DISK ENCLOSED™

# AC's TECH /AMIGA

*For The Commodore*

# DO-IT-YOURSELF HARD DRIVES

HARD DRIVE GARAGE

**Programming the Amiga in Assembler**

**Implementing an ARexx Interface in C**

**Programming the Amiga's GUI in C—Part IV**

**CAD Application Design—Part III**

**Low-Level Disk Access**

**Designing a Ray Tracer—Part II**

**AudioProbe in Modula-2**

**Writing MusicX Protocols**

*Plus*

## Build Your Own
## MIDI Interface

# Contents
Volume 2, Number 1

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

**Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:**

AC's TECH Submissions
PiM Publications, Inc.
One Currant Place
Fall River, MA 02722

# Startup-Sequence

## It's Official—AmigaDOS 2.04!

Move over AmigaDOS 1.3...AmigaDOS 2.04 is officially here! The OS code has been frozen. The final release version of AmigaDOS release 2 is 2.04 with Kickstart v37.175, and Workbench v37.67. These are the OS versions that are being put into ROM in the new machines and the Enhancer Kit (upgrade for the A500/2000).

## Easy Upgrades

Upgrading Amiga 500s and 2000s will be accomplished via the 2.04 Enhancer Kit, which will be available through the authorized dealer channel. This will be a ROM and software upgrade. Once the new 2.04 ROMs are in, say goodbye to AmigaDOS 1.3, forever. However, there are several third parties that are developing, or have developed ROM Towers (switches) for the A500/2000. These ROM Towers hold both 1.3 and 2.04 KickStart ROMs and allow selection of the boot ROM via an externally mounted switch. Add a little intelligence to your startup sequence, and presto! Dual personality! Any side effects? Hmmm... It's too early to say, but there is at least one potential problem: AmigaDOS 1.3 does not recognize AmigaDOS 2.04 hard links. These will look like empty files under 1.3. If anything, these will cause a little (or lots) of confusion. Just be cautious...

## A3000—I Want My ROM

What about an upgrade for the original Amiga 3000 with the SuperKickstart? Sources at Commodore say that a *free* 2.04 upgrade will be made available through the dealer channel. This will be a five disk *software* upgrade (3000 Install, Workbench, Extras, AmigaFonts, and KickStart.) Dealers are authorized to charge a nominal copying fee for the disks. Will there be a 2.04 ROM upgrade/system trade-in for the SuperKickstart Amiga 3000s? At press time the details are unspecified, but there is a plan on the table. Ask you dealer for more specific information.

What about running AmigaDOS 1.3 on an A3000 with 2.04 in ROM? You can't...right now. The A3000 uses two 256K ROMs as opposed to a single 512K ROM on the A500/2000. It's this reason, plus a hundred other engineering and legal reasons, that you won't see an A3000 ROM Tower switch. However, I'm sure that some bright entrepreneur/developer will come up with a way (i.e. marketable product) to boot AmigaDOS 1.3 on an A3000 with 2.04 in ROM.

What's your best bet on upgrading a SuperKickstart A3000 to AmigaDOS 2.04? If you *have* to run AmigaDOS 1.3, then stay with a software upgrade—at least until there is another way to run 1.3 on a 2.04 machine. If you are a developer, then its probably a good idea to stay with the SuperKickstart so you can continue to easily get and install system software upgrades. Go with the hardware upgrade if you need to use a 68040 CPU in your A3000. The new ROMs fixed cache bugs and added OS cache calls. Also, it is rumored that some SCSI devices do not like to see the old 1.4B5 ROMs. In these cases, the 2.04 ROM are required. Take your pick.
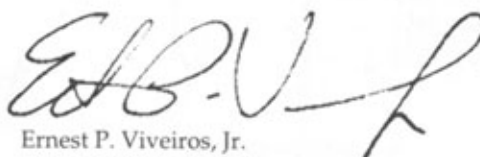
Finally, starting with AmigaDOS 2.04, all new A3000s will come with AmigaDOS 2.04 in ROM.

*Looking to the future...*

## Next step AmigaStep?

One of the better applications bundled with the powerful UNIX-based NeXT™ workstations is its extremely powerful application-builder NeXTStep™. NeXTStep is an extremely intuitive, easy-to-use, yet incredibly powerful GUI/application builder, which puts the power of real programming into the hands of the power user. NeXTStep is not an everyman's solution to programming, but it does open up programming to the advanced user, who has the mindset to program, but can't get over the complexities of programming the GUI at the medium-to-low level. NexTStep can also be used in a lower-level capacity to develop commercial-quality software.

Rumor has it that Commodore is actively working on an application builder that will be NeXTStep and more! This versatile and powerful application builder will be able to handle all aspects of the Amiga GUI, ARexx, IDCMP, font sensitivity, and more. When can we expect to see this programming marvel? No one's talking, but a good guess says that they'll have it in beta sometime during the first quarter of '92. I can hardly wait...

Ernest P. Viveiros, Jr.
Technical Coordinator

# Spartan
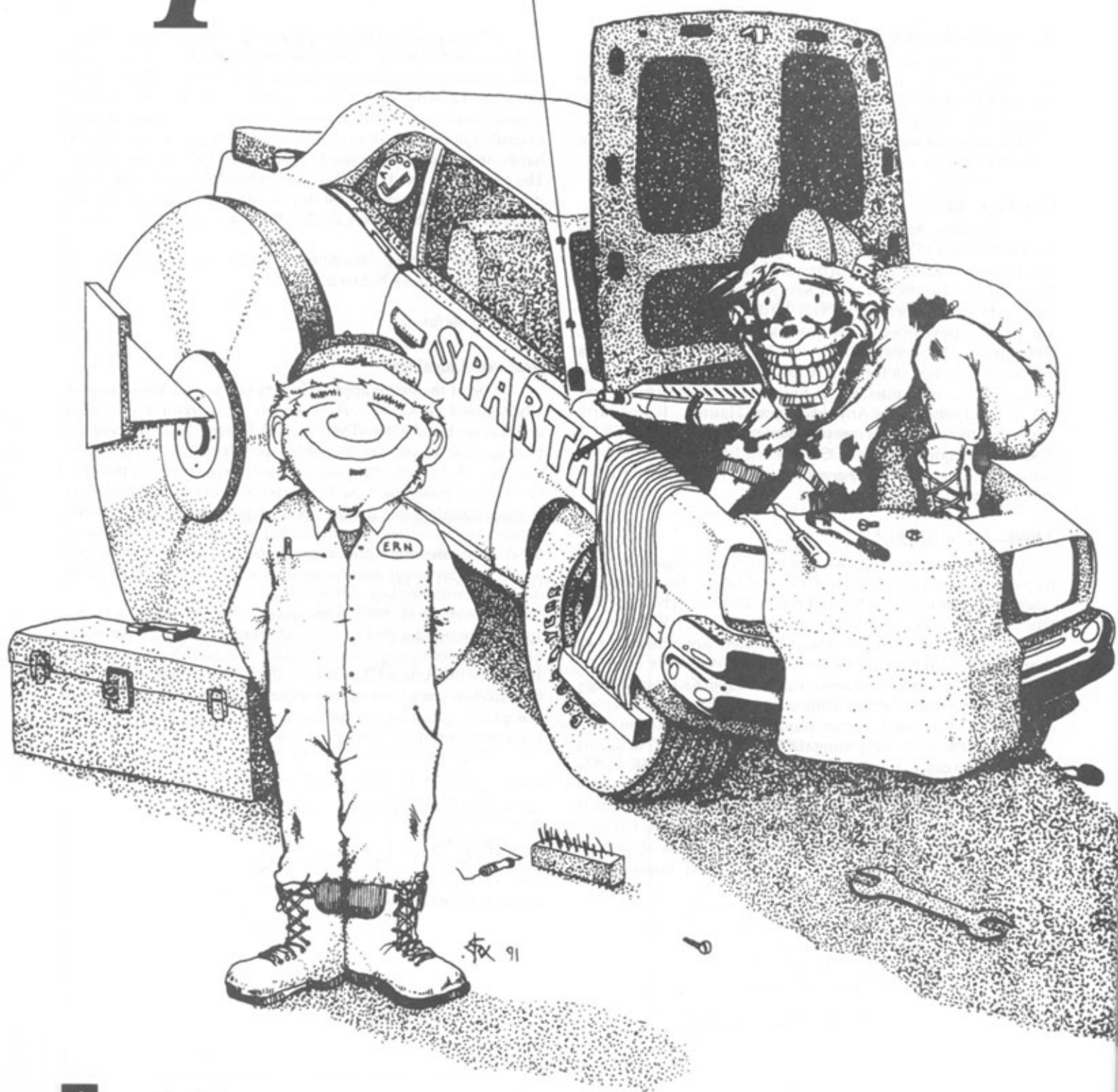
# Build A SCSI Interface for Your Amiga 500/1000

## by Paul Harker

*CAUTION: Be careful when attempting to build and connect hardware projects to your computer. Always check your work twice before attaching the project to your computer. Attaching a home-built project to your Amiga may void its warranty. PiM Publications, Inc, its agents, or the author, is not responsible for any resulting damages from this project. As always, use care and common sense.*

If you are like many Amiga users, you spend entirely too much time swapping disks and get bored waiting for your icons to display and your programs to load. Perhaps it is time to add a hard drive. The first step in adding a hard drive is the interface between the computer and the drive. For the Amiga, the defacto standard is SCSI (Small Computer System Interface). Spartan is a build-it-yourself SCSI hard drive interface for the Amiga 500 and 1000. Using CPU (non-DMA) data transfers, Spartan is capable of data transfer rates of well over 300K per second—more than adequate for the average Amiga user. Spartan is 100% compatible with all Amiga software, including popular hard drive utilities such as Quarterback.

Spartan supports all standard SCSI hard drives, removable media SCSI drives such as the SyQuest, and even the old IBM type ST-506/412 drives (see Adaptec Sidebar). Drives can be assigned to seven different SCSI addresses, and up to fourteen hard drives can be controlled.

Physically, the Spartan SCSI interface consists of a small circuit board connected to the expansion bus of the Amiga. Using the powerful AM5380 SCSI control IC, two standard TTL chips, and a few capacitors and resistors, the interface gives your Amiga the ability to communicate with standard SCSI drives. The interface address in memory is selectable to avoid conflict with expansion memory or other add-on hardware. To remain inexpensive and to keep the circuit simple, the Spartan interface does not support autoconfiguration or autobooting.

The Spartan device driver is a tiny 2048-byte file which is loaded via the MOUNT command, and a properly edited mountlist. Low-level format routines are provided for SCSI drives, and for Adaptec controllers. Also included is the 'C' source code the the low-level format routines to allow modification for support of non-standard SCSI drives, or control of other types of SCSI devices.

## Addressing

The Spartan interface is what is known as a memory mapped device. This deviates from the AutoConfig method utilized by many devices on the Amiga. AutoConfig looks for an unused area in memory, and then assigns that address to the hardware. Due to the simplicity of the Spartan circuit design, we must do the job of locating a safe area in memory ourselves. As Spartan decodes memory in 64K blocks, the safe area to which the interface is addressed will be refered to by it's "base address." The base address is the high byte of the actual address. As an example, if the address is 2FFFFF, then the base address is 2F.

There are two areas in the Amiga's memory map which are good candidates for locating Spartan. First, the area from F00000 to F80000 is unused by the Amiga operating system and makes F7 (address F70000) a perfect base address for your interface. However, if you have an Amiga 500 expansion memory cartridge such as AdRam 540 or similar units with more than 500K of expansion, you may not be able to use this address. This is because they remap this area for expansion RAM. The normal one half MB A501 or A501 clone expansions will NOT interfere with this address. If you are in doubt, refer the the documentation that came with your expansion as to where it installs memory.

If you are unable to locate your interface at base address F7, then you must locate it between 200000 and 8FFFFF. This area is normally used for expansion memory, however, unless you have a full 8MB, there will be room to address the interface. The documentation which came with your memory expansion should allow you to locate an unused area within this range. For example, if your expansion memory is:

| Start | | End |
|-------|-----|-------|
| 200000 | to | 27FFF |
| 400000 | to | 47FFF |
| 800000 | to | 87FFF |

then a base address from 28 to 3F and from 48 to 7F would be valid.

Regardless of whether your base address is F7 or elsewhere in the memory map, note the address, as it will be used in configuring both your hardware and software.

### Building the Interface

By far the easiest and most reliable method of building the interface is to obtain the etched and drilled circuit board available from the author. If you wish to etch your own board, the pattern is provided in Figure 1. If you wish to build using another method of construction, see Figure 2 for the schematic. A carefully built wire wrapped circuit should have no problems if you keep wire lengths to a reasonable minimum. The following assembly discussion refers to the etched circuit board, but should be helpful for any type construction.

Refering to the placement guide (Figure 3) note that all parts on the interface are inserted from the component side of the board. That is, the side WITHOUT the copper traces. Insert and carefully solder the three IC sockets in place. If you examine the sockets, you will notice one end has a small notch. Be sure this notch is at the pin 1 end as indicated in Figure 3.

Insert and solder the resistor packs (RP1, RP2, RP3, RP4) as indicated, again paying attention to the orientation of pin 1. Pin 1 is indicated by a stripe or dot on the part. Insert and solder C1, C2, and C3 as indicated, and clip the leads neatly. Note the polarity of C4 on Figure 3. The capacitor itself has one lead marked + on the side of the capacitor. Insure that this lead is inserted as indicated. Solder and clip the leads.

Install nine jumpers as indicated using either your left over capacitor leads, or small lengths of wire. Solder and neatly trim the jumpers. Next, insert and solder the 50-pin SCSI connector.

The 86-pin connector to the Amiga poses a bit of a problem. For simplicity of design, the board is constructed so that the connector slot is parallel to the board. Normally this requires an 86 pin right angle connector. That is, the pins exiting from the back of the connector make a right angle turn. However these connectors are not available in small quantities. If you are lucky enough to locate one, simply solder it in place. Otherwise you can modify the connector listed on the parts list to do the job nicely. Refering to Figure 4, carefully bend one row of pins on the connecter at a right angle. Insert this bent row of pins into the row of holes CLOSEST to the edge of the board. You may wish to put a drop or two of super glue to fix the connector in place on the board. Once in place, solder the row of pins. Now, using short lenghts of wire, such as resistor leads, connect the remaining pins of the connector the the corresponding hole as shown in the drawing. Solder in place both at the pin, and on the circuit board.

Install the dip switch as indicated. The dip switches are for setting the base address of your interface. Refering to Figure 5, you can see that each switch refers to a bit in the base address. A closed switch is a 0, and an open switch is a 1. Several example settings are shown. Just note that the high bit is the switch closest to the SCSI connector and the low bit is the switch closest to the Amiga connector. Set the switches to agree with your base address.

Check your parts placement, and soldering. Insure that you have no solder "bridges". All it takes is one 'OOPS' and you could damage your interface or computer. Once you are done checking your work, have someone else double check it for you. If all checks OK, proceed with inserting the IC's.

NOTE: IC's are STATIC SENSITIVE. Use proper measures to prevent damage. If you are unsure of how to do this, please have someone who is, insert your IC's for you.
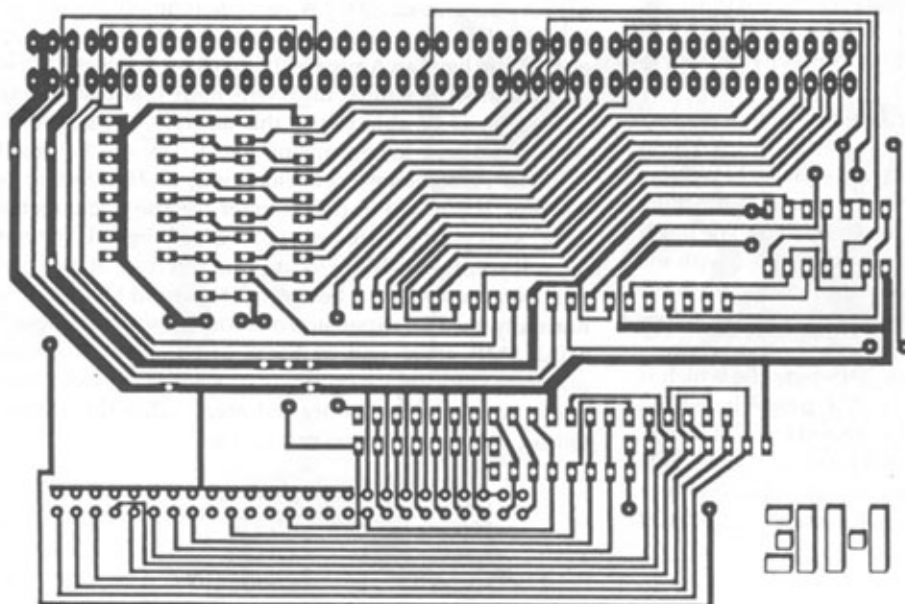


**Figure One**
The Spartan PCB Layout

Figure Two: The Spartan Schematic Layout

ALL CAPACITANCES IN MICROFARADS
C1-C3 CERAMIC DISK
C4 TANTALUM

SPARTAN
SCSI INTERFACE
(C) 1991 Paul Harker

All Odd Numbered
SCSI Pins to
Ground

Address Jumpers

1K Pullup    RP4

74LS00

74LS688
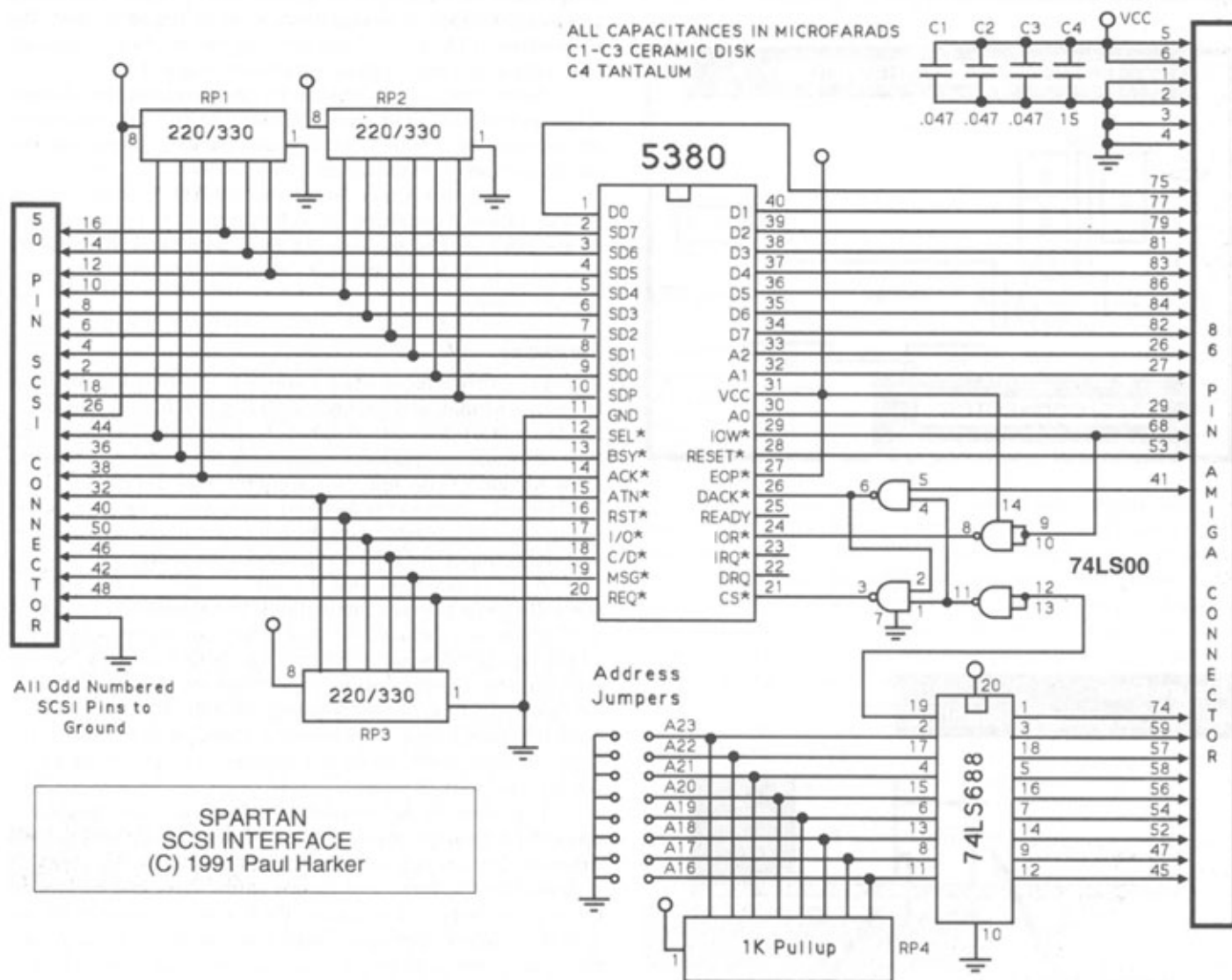
Insert the three IC's in their respective sockets, insuring that pin one is correctly positioned. The pin one end can be identified by either a notch at the end, or a dot next to pin one. This pin one should be at the pin one end of the socket. This completes the assembly of your Spartan interface!

### Assembling the SCSI System

In addition to the Spartan interface and software you will require the following:

SCSI Hard Drive (or Adaptec 40X0/ST506-412 combination)
50-Pin SCSI Cable
IBM PC/XT type power supply
Power cable for power supply
Enclosure for drive and power supply

An IBM 'clone' case will accept both the power supply and the hard drives nicely, and are available inexpensively from many sources. While powering a small hard drive directly off the power supply of the Amiga 1000 is possible, I don't recommend it and it is absolutely prohibited with the minimal power supply of the 500.

86 PIN AMIGA CONNECTOR

C4 · RP4 · SW1 · 74HCT688 · C3 · 74LS00 · C1 · C2 · AM5380 · RP1 · RP2 · RP3

SCSI CONNECTOR

Component Side

Use Short Length of Wire and Solder at Both Ends.

Bend At Right Angle and Solder.
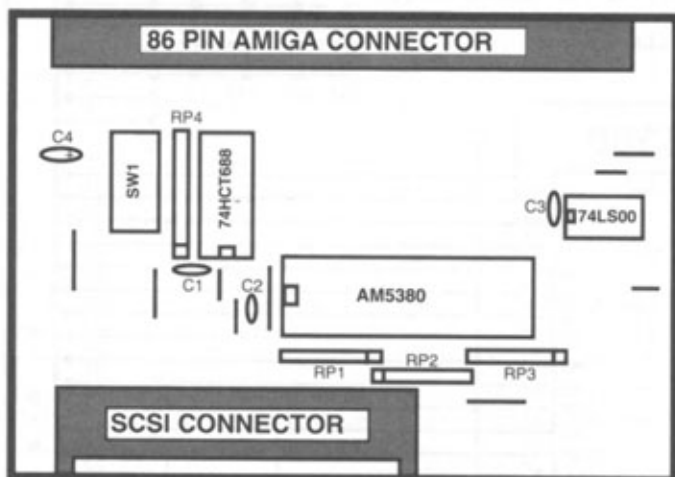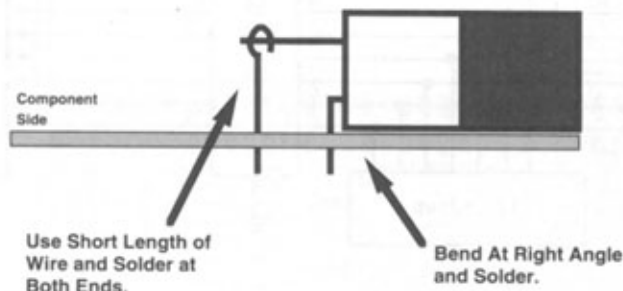
Mount your drive and power supply in your enclosure, insuring that the circuitry does not short to the case or to another component. Layout is not important. Plug a power connector from your power supply into the power socket on the hard drive. Set the SCSI address of your drive to 0 as directed by the owners manual.

Ensure that the Amiga is turned off and connect the assembled Spartan interface to the expansion bus. On the 500, simply remove the snap on cover from the left end, and firmly push the interface in place (components facing up if using the etched board). On the 1000, the interface installs on the right side of the computer (once again, component side up).

At this time, take a deep breath, and power up your Amiga. If it does not boot up properly, IMMEDIATELY turn the power off and recheck all your work. If you continue to have problems, see the section on PROBLEMS.

Turn off your Amiga. Install the 50 pin SCSI cable between the interface and your hard drive. Insure that pin 1 of the cable is at the pin 1 end of the connectors at both ends. Recheck all connections. Once you are sure your cabling is correct, proceed with formatting.

## Formatting

Hard drives require two forms of formatting, the low-level (physical) format, and the high-level (logical) format. The high-level format is what you are already familiar with. That is, it is the type of format you perform when using the 'Format' command from AmigaDOS or 'Initialize' from the Workbench. The low-level format, which is only used on hard drives, marks bad sectors of the drive as unusable, sets the interleave (more on this later) and stores important information about the configuration.

A set of low-level format utilities, AFormat, for Adaptec controller/ST506 drive combinations, and SFormat, for standard SCSI drives, are included on disk. These are executed from the CLI by typing 'AFormat' or 'SFormat' at the DOS prompt. You are then presented with a series of questions about your particular configuration. Once these are answered, it then performs a complete low-level format of your drive. These programs tie up the SCSI bus completely, so do not attempt any other SCSI access while performing a low-level format, or you will meet the GURU.

To perform the low-level format, first boot your Amiga, and open a CLI. Turn on the power supply to the hard drive. The hard drive will go through some initialization activity. This may be indicated by the flashing of the led on the drive and controller. Once this activity is finished, run the low-level format utility for your drive. You will be prompted for information regarding your particular configuration. AFormat requires much more information than does SFormat. You need to know several things about your hard drive to perform a low-level format. This information should have been included with the drive. A text file on disk, 'DriveSpecs', gives the needed information for many popular hard drives. If you can't find information for your drive, contact the drive manufacturer.

AFormat will prompt you for:

| | |
|---|---|
| Base address | Base address in HEX |
| Adaptec Card Type | Enter 1 for the 4000(A), 2 for the 4070. |
| SCSI Address | Enter the drives address |
| Drive # | Drive 0 connects to J0 and Drive 1 connects to J1 (refer to the Adaptec Manual). |
| Number of Cylinders | Enter the number of cylinders. |
| Number of Heads | Enter the number of heads |
| Step rate | Enter the step rate of your drive. |
| Interleave | Enter 2 |
| Reduced Write Current | Enter 'Y' or 'N' |
| Enter Cylinder # | Enter the starting cylinder number for Reduced Write Current. |

AFormat will then prompt for information regarding media errors on your hard drive. This information is normally noted on a label attached to the drive. Enter this information as requested. To terminate error entry, enter a 0 (zero) at the 'Cylinder:' prompt.

SFormat is much simpler to use and only prompts for base address, SCSI address, and the interleave:

Base address.       Base address in HEX
SCSI Address        Enter the drives address.
Interleave          Enter 2.

Following data entry, you will be given a last chance to bail out. If you continue, the drive will become active for about 5 minutes. (20MB drive, correspondingly longer on larger drives).

If you have an non-standard SCSI hard drive which will not format with SFormat, it will work with your Spartan interface if it has been low-level formatted. The drive can be formatted on another computer as long as the bytes-per-block is 512, in fact most SCSI drives are sold with a 512 byte-per-block low level format already on them. The Macintosh and Amiga both use this block size, so you can format your drive (low-level) on another computer, then hook it up to your Amiga to perform the DOS format.

For those with C language experience, the source code for SFormat is included on disk. This can be easily modified to conform to the low level format command sets used by non-standard SCSI drives. This source is compileable by Aztec C.

Make a copy of your AmigaDOS 1.3 Workbench. This will be refered to as your boot disk. The copy of spartan.device on disk is addressed to base address F7. If your Spartan interface is at base address F7, simply copy spartan.device from the DEVICE directory into the devs: directory of your boot disk. However, if you have a base address other than F7, you must use the supplied utilty, SetBase, to modify the base address of spartan.device.

The usage of SetBase is:

SetBase <filename> <BaseAddress>

Filename will normally be 'spartan.device' (including the path), and BaseAddress is the two character hex base address of your interface. SetBase will create a new file named '<filename>.XX' where XX is the base address of the driver. Copy this new device to the DEVS: directory of your boot disk and rename it spartan.device.

Edit your mountlist to include the following (modified to your configuration). Please note that unlike most AmigaDOS commands, the MOUNT command is case sensitive. Be sure that the device file name and mountlist device entry match exactly.

DH0:

| | |
|---|---|
| Device = spartan.device | |
| Unit = 0 | ;See Text. |
| Flags = 0 | ;not used |
| Surfaces = 4 | ;Number of heads on ;drive |
| BlocksPerTrack = 17 | ;See Text |
| Reserved = 2 | ;Don't change |
| LowCyl = 0 ; | |
| HighCyl = 614 | ;Bottom and top ;cylinder numbers; |
| Buffers = 30 | ;Refer to AmigaDOS ;manual |
| BufMemType = 0 | ;Refer to AmigaDOS ;manual |
| MaxTransfer = 131071 | ;Don't change |
| Stacksize = 4000 | ;Don't change |
| GlobVec = -1 | ;Don't change |
| FileSystem = L:FastFileSystem | ;Don't change |
| DosType = 0x444f5301 | ;Don't change |

A copy of this file is on disk. Some entries require explanation. Enter 0 as the unit number for your first hard drive. Unit numbers for additional drives are discussed in the 'Multiple Drives' section of this article. For an intial setup enter 0 for LowCyl and enter the number of cylinders, minus one, for HighCyl. The interleave in the mountlist is ignored by Spartan, as your interleave has already been set by the low-level format. Enter the blocks-per-track of your SCSI drive (17 is usual), or if using an Adaptec controller enter 18 for the 4000(a) or 26 for the 4070.

Once this is complete, edit the Startup-Sequence of your boot disk and add 'Mount DH0:' as the first command. Reboot your Amiga using your boot disk. From the CLI enter:

Format drive dh0: name MyDriveName quick

**Figure Five**
SW1 Base Address Settings

## SW1



```
Closed Switch = 0
Open Switch = 1
```

**Example Base Addresses**

```
7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 = F7
0 0 1 0 0 0 0 0 = 20
0 1 1 0 1 1 1 1 = 6F
```

You may omit the quick option. However, unlike a floppy there is no need to write to each track. Your formatting will just take longer. You now have a fully useable hard drive which can be accessed like any other DOS drive!

NOTE: The AmigaDOS version 1.3.2 Format command has a small problem with Spartan. After formatting with the 1.3.2 Format command, you will get a 'Format Failed' message, and AmigaDOS will report 'Not a DOS Disk' if you try to access the drive. Reboot, and AmigaDOS will recognize the disk as being a properly formatted DOS disk. The older 1.2 and 1.3 versions of Format work without this problem.

## Optimizing Performance

As you recall, when you performed your low-level format, you needed to enter the interleave. Interleave is the method that is used to store data on you hard drive in a manner best suited to your particular system.

This discussion of interleave will focus mainly on reads, but the same information pertains to writes as well. As a hard drive platter spins, the data is accessed one block (512 bytes) at a time, and then sent to the computer. As the platter advances to the next block, often the computer is not quite ready to receive more data, and the platter has to make a full revolution before the heads are

again in position to read the block. However, this full revolution is much more time than the computer needed to be ready, and much time is wasted. This wasted time equates to low data tranfer speeds. What interleave does is alter the order in which blocks are read and written. On a hard drive without interleave (referred to as an interleave of 1), the data blocks are read/written sequentially:

|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|17|

But with an interleave, they are 'shuffled' evenly, as shown in this example of an interleave of three:

|01|12|07|02|13|08|03|14|09|04|15|10|05|16|11|06|17|

As you can see, block 01 is three blocks from block 02 (thus an interleave of three), so that the order of access is read one, skip two, read the next, skip two, etc. The time spent skipping blocks allows the computer to catch up and be ready for data when the drive is ready to send it. If the interleave is too large, the drive skips blocks while the computer waits, and if the interleave is too small, the drive is ready before the computer and must make an additional full revolution.

What we want to do is select an interleave for the hard drive to get the best performance. The disk performance tester DPerf2 will help in this task. The author was kind enough to allow this program to be included with Spartan. Please take the time out now to read his documentation (on disk).

Run Dperf2 on your hard drive and write down the results. Re-format your drive (both low-level and DOS) with various interleaves to see what gives your system best performance. An interleave of 2 gives best performance with many drives.

Be careful of one bug in DPerf2. If for some reason it is unable to access the specified drive, it writes all over DF0:, so be sure to write protect the disk in DF0:!

### Adaptec Users—

*To use an interleave of 1, you must edit your mountlist to reflect one less BlocksPerTrack. For example a 4000(A) controller will now be BlocksPerTrack=17, and a 4070 will be BlocksPerTrack=25. You must reboot the Amiga, and 'MOUNT DH0:' for this to take effect. To go back to an interleave other than 1, restore your mountlist BlocksPerTrack, and reboot. Note that you lose your extra 10% storage with an interleave of 1. Refer to the Adaptec Manual for details.*

### All Drive Types—

*The sequence for optimizing interleave:*

*AFormat or SFormat*
*DOS Format*
*Dperf2*
*repeat...*    *(Adaptec users: edit mountlist and reboot if going to/from an interleave of 1)*

# The Adaptec Controllers

Adaptec produces a line of hard drive controllers which allow standard IBM type (ST50612) drives to communicate as SCSI drives. Each controller supports up to two hard drives. The combined cost of one of these boards (as low as $80 from some suppliers) and a surplus or used IBM type drive is often an economical way of adding drives to Spartan. The Spartan software supports the following Adaptec controllers:

4000    The original MFM model
4000A   The standard MFM model
4070    Supports RLL drives

The Adaptec 4070 is an RLL (Run Length Limited) controller. RLL is a method of string 50% more data on a hard drive than traditional methods. For example, a 20meg drive formats to 30meg using RLL. However, to make use of RLL encoding, an RLL certified drive must be used. Non-RLL (MFM) hard drives can be controlled with the Adaptec 4000 or 4000A.

One great feature of the Adaptec controllers is an unique method of storing data. Due to this method of data storage, each drive formats to 10% LARGER than normal. So a 20meg MFM drive would format to 22meg, and a 20meg RLL drive would format to 33meg!

To use an Adaptec controller with Spartan you would require the following:

> Adaptec Controller with User's Manual
> (4000, 4000A or 4070)
> ST50612 type Hard Drive
> 34 Pin Card Edge Ribbon Cable
> 20 Pin Card to Socket Ribbon Cable

The user's manual does NOT come with the controller and has to be ordered. Refer to this manual for any questions you have regarding interfacing your drive to the Adaptec controller.

## Multiple Drive Configurations

Spartan supports seven SCSI devices. Using Adaptec drives, this allows a system containing up to 14 hard drives! Although it is seriously doubtful that anyone would take it that far, it is possible. The following section details the information needed to attach additional drives to the Spartan system.

Adaptec controller users can add a second drive (drive 1) to the controller as described in the Adaptec manual. Be sure to address your second drive as drive 1, and remove the terminator resistor from drive 0.

Adding additional SCSI drives is not difficult. You need a SCSI cable with additional connectors to daisy-chain as many drives as you are using. Remove the terminator resistors from all but the last drive on the SCSI cable. Also, be sure your power supply can handle the load of the extra drives.

To communicate with drives, you need to set the SCSI address of the drive to an unused address between 0 and 6. Refer to your drive manual for details on how to configure your drive's SCSI address. Address 7 is reserved by Spartan. Use this SCSI address when performing the low-level format on this drive.

Each drive must have it's own mountlist entry to be used as an AmigaDOS drive. The unit number in the mountlist tells Spartan at what SCSI address to find the drive. The following chart demonstrates this.

| MountList Unit # Assignments | | |
|---|---|---|
| SCSI | Unit# Drive 0 | Unit# Drive 1 |
| 0 | 0 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | 5 |
| 3 | 6 | 7 |
| 4 | 8 | 9 |
| 5 | 10 | 11 |
| 6 | 12 | 13 |
| 7 | Reserved for Spartan | |

Only Adaptec controllers with a second drive use the drive 1 unit numbers. All other SCSI drives use the Drive 0 numbers.

So, as an example, if you set the unit# in your mountlist to 6, Spartan will interpret this as drive 0 at SCSI address 3.

Once a drive has been connected to the system, and a mountlist prepared, the drive must be low-level formatted, mounted and a DOS format performed. The drive will then be ready for use.

## Partitioning Your Drive

Unlike floppy disks, hard drives can be partitioned. What this means, is that one large hard drive can be made to act like two or more smaller ones. This is especially useful on very large drives to prevent file fragmentation, and speed disk access.

To partition a drive, you must again edit the mountlist. Each partition requires a separate mountlist entry. For example, you could partition a 20MB drive as one 10MB drive, and two 5MB drives. Lets call our 10MB partition DH0:, our first 5MB partition DH1: and our second 5MB partition DH2:.

For the purpose of our example lets assume our 20MB drive has 600 cylinders. If we were to mount this as a 20MB drive we would have a mountlist entry with the following line:

LoCyl=0, HiCyl=599

To partition the drive we would first create mountlist entries for all three partitions (DH0: DH1: and DH2:). The mountlist entry for DH0: we would edit the cylinders as:

LoCyl=0, HiCyl=299

This gives our DH0: access to the first half of the drive. For drive DH1: and DH2: we would edit the mountlist entries as follows:

LoCyl=300, HiCyl=449    ;(DH1)
LoCyl=450, HiCyl=599    ;(DH2)

This gives both DH1: and DH2: one quarter of our hard drive, or 5MB each. Be sure that your partition's cylinders don't overlap. A complete example mountlist is on the disk with the filename Partition.example. Remember that each partition requires it's own mountlist entry and device name.

The drive must be low-level formatted, however separate low-level formats are not needed or possible for partitions. Each partition must be be MOUNTed and separately formatted with the AmigaDOS format command.

## Booting your Hard Drive

Although Spartan does not autoboot, it can be easily used as a system disk by means of your boot disk. First, copy the entire contents of an original Amiga Workbench 1.3 to your hard drive. Replace the s:startup-sequence on your boot disk with the startup-sequence from the Spartan archive. You may want to copy 'disk.info', also. (I'm no artist, but I like it!). What this startup-sequence does is simply mount DH0:, ASSIGN all system directories to DH0:, then EXECUTE the startup-sequence in S: (now assigned as DH0:S).

To boot your system, power up your computer. Insert Kickstart if you have a 1000. Turn on your hard drive, and wait for initialization activity to cease. Now insert your boot disk, and everything else takes care of itself. Once the system is running, the boot disk is no longer needed.

## Notes

Amiga 500 owners may wish to leave the boot disk in the internal drive at all times, and use a single switch to power both computer and hard drive. However, if the hard drive has not finished it's intialization when the startup-sequence starts accessing it, the system will lock up. To avoid this, insert an appropriate length WAIT command immediately after the MOUNT command in your startup-sequence.

The 5380 SCSI controller I.C. is manufactured by several vendors, including AMD, LOGIC Devices, and NCR. I have tested the AMD and the LOGIC Devices versions, and of the two only the AMD device is compatible with the Spartan software.

Some ST506/412 drives have errors mapped on sections of the drive that Adaptec controllers do not format. These errors will cause AFormat to fail. Do not enter any errors mapped in a cylinder higher than your high cylinder, and any errors with a bytes-from-index higher than:

|                     | 4000(A) | 4070  |
|---------------------|---------|-------|
| Interleave of 1:    | 9792    | 9817  |
| Other Interleaves:  | 10188   | 10036 |

The most likely causes of a newly constructed interface to not work, are solder bridges and bad solder joints. Clean your board's foil side with isopropyl alcohol to make it easier to see any problems.

Check that your cables are all in good shape and are properly oriented. Inspect the pin one orientation of all IC's and resistor packs (RP1,2,3). If an IC was installed backwards, and the power applied, it was most likely destroyed.

Check that C4 is installed with the + as indicated in the drawing. If you put this in wrong, a small puff of smoke from it will probably let you know. If this happens, it can be replaced with a tantalum capacitor with a value of 15 to 22 uF at 6 volts or more.

Another likely problem is improper addressing of either the interface or of spartan.device. Be sure you have selected a valid base address for your system. Check that you have properly set the dip switches on on your interface and that you have set the base address of spartan.device using SetBase.

There are two simple troubleshooting programs in the Low-Level directory, ATest, and STest. These programs are paired down versions of AFormat and SFormat, which will not perform a format, just establish communication with the drive.

## Conclusion

I sincerely hope that Spartan opens up your world of computing as much as it has for me. An Amiga with a hard drive is so much more powerful and versatile, that it's like using a completely new machine.

The etched and drilled circuit board and the hard to find AM5380 are available from the author. The prices are as follows:

Etched and Drilled Circuit Board    $20.00
AM5380 SCSI Controller              $15.00

Add $1.50 shipping for each order. Write for quantity discounts on orders of five or more. Send check or money order to: Harker Electronics, 255 Valley N.W., Grand Rapids, MI 49504

I am unable to answer questions about Spartan by phone, but can be contacted by mail or via GEnie P.Harker, or CompuServe 71546,1665.

# CAD
# APPLICATION DESIGN—PART 3
# OBJECT-ORIENTED CAD

BY FOREST W. ARNOLD

## Introduction

C++, SmallTalk, Eifel, Flavors, objects, classes, BOOPSI...OOPS! Nowadays, just about every software magazine you pick up has a reference to at least one of these programming languages or terms. It seems as though programmers all over the world have jumped on the bandwagon of object-oriented programming and are busily creating objects. Since we don't want to be left behind eating technological dust, we're also going to jump on board! We are going to develop an architecture for implementing geometric objects as true object-oriented objects, and develop several "classes" of objects for our mini-CAD program—and without an object-oriented language, just plain old ANSI C and some programming and software packaging tricks! But first, we need to take one more look at geometric transforms, and then see how to interactively create new graphical objects using the event-driven programming methods we developed last time (*AC's Tech*, Vol 1.3).

## Geometric Transforms Revisited

In Part 1 of this series, we discussed the geometric transforms used to translate (move), rotate, and scale (resize) geometric objects modeled with vectors, which are just point coordinates. Translations are the simplest transforms: we simply add a vector (x,y coordinate pair) to all the coordinates defining a model object. Scaling and rotation transformations are more complicated, since both of these transformations require "reference points." When an object is scaled, all the coordinates in the object either move away from the reference point or move toward the point, and when an object is rotated, all its coordinates move in a circle around the reference point. To correctly rotate or scale an object, we first translate the object's coordinates so they are relative to the reference point, rotate or scale the object, then translate it back to its original location.

How do we choose a scale or rotation reference point? One way is to have the CAD user make the decision by picking a reference point. Another way is to use the initial "pick point" as the reference point, and still another way is to use an arbitrary point. This is the technique we used in Part 2 of this series. Objects were rotated and scaled using the centers of their bounding boxes as the transform reference points. We will again use an arbitrary reference point to transform the objects in our mini-CAD program. However, instead of always using the center of an object's bounding box, the reference point will depend on the transform being applied. For rotations, the reference point will be the center of the object's bounding box; and for scaling transforms, the reference point will be the corner of the bounding box which is diagonally opposite the corner closest to the pick point on the object.

Another common transform provided in most CAD systems is "flipping" objects. In this version of our mini-CAD program, we will add a "flip" menu item to our "Action" menu. Objects are flipped by interchanging their tops with their bottoms, or by interchanging their right- and left-hand sides. As an example, suppose we have a triangle whose coordinates are:

| x1 | y1 | x2 | y2 | x3 | y3 |
|----|----|----|----|----|----|
| 0  | 0  | 5  | 0  | 5  | 3  |

Figure 1 shows how this triangle will look after being flipped horizontally around the point (0,0) and vertically around the point (5,3). The transform to flip an object is just a scaling transform with negative scale values. Since objects are flipped with scaling transforms, a flip reference point is needed. In our mini-CAD program, we will use the pick point on the object as its flip reference point. The scale matrix to flip our example triangle horizontally around the point (0,0) is:

$$\begin{vmatrix} -1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{vmatrix}$$

And to flip it vertically around the point (5,3), the overall transform matrix is formed by the following sequence of matrix multiplications (in our program, we call our transform library matrix procedures to do this work):

| Translate to original location | | | | Scale | | | | Translate to flip reference point | | |
|------|------|------|---|------|------|------|---|------|------|------|
| 1.0 | 0.0 | -5.0 | | 1.0 | 0.0 | -5.0 | | 1.0 | 0.0 | 5.0 |
| 0.0 | 1.0 | -3.0 | X | 0.0 | -1.0 | 1.0 | X | 0.0 | 1.0 | 3.0 |
| 0.0 | 0.0 | 1.0 | | 1.0 | 0.0 | 1.0 | | 0.0 | 0.0 | 1.0 |

As Figure 1 shows, after an object is flipped, it will not be in its original location, unless it is flipped around its center. Some CAD systems automatically move flipped objects back to their original locations.

## New Program Commands

A CAD program which can transform objects is not very useful unless it can also create objects, so we are going to add object creation procedures to our program. The new procedures will enable us to interactively add rectangular shapes, polylines, and polygons to our model world. We'll also add procedures to delete objects, to view and modify an object's values using a

**Y Axis**

Triangle flipped vertically around (5,3)

(0,5)

Triangle flipped horizontally around (0,0)

(5,3)

Original Triangle

(-5,0)

(5,0)

**X Axis**

**Object Structure**

| Class | Next | ... | ... |

**Class Structure**

| | **static procedures** |
|---|---|
| Super Class | |
| Object Size | |
| Setup | → setup() |
| initialize | → initialize() |
| deallocate | → deallocate() |
| setvalue | → setvalue() |
| getvalue | → getvalue() |
| update | → update() |
| display | → display() |
| erase | → erase() |
| ... | → ... |
| extent | → extent() |
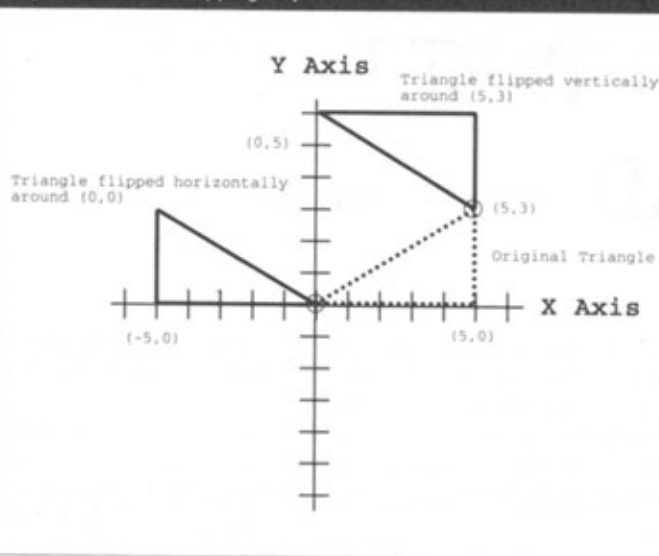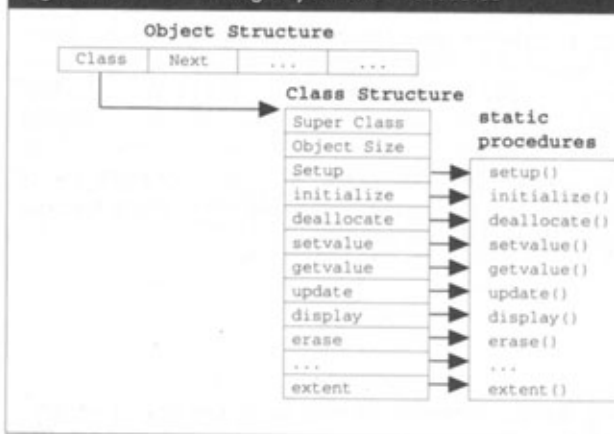
"Show/Edit" requester, to flip objects, and to move the individual points in a line or polygon. The following paragraphs describe the program commands associated with these procedures and the user actions for executing them.

Objects are added to a drawing window by first picking an "Insert" menu item, followed by a menu subitem for the type of object to insert. The menu item-subitem choices are "Insert-Shape," "Insert-Line," and "Insert-Polygon." Shapes are rectangles, lines are a series of two or more points connected by line segments, and polygons are lines whose first and last points coincide. Thus, polygons are just closed figures constructed with points and line segments. After the "Insert" menu pick is complete, the object is added by picking its starting coordinates, dragging the cursor to locate another set of coordinates, and then "clicking" the select button. For shapes, the two sets of coordinates define two of the rectangle's corners, and the shape insert action is complete. For lines and polygons, the coordinates define the first line segment. To continue adding points to lines and polygons, one drags to another location and presses the select

button to add the new point. The line and polygon insert action is completed by "double-clicking" the select button at the location desired for the last point. If the object being inserted is a polygon, the program will automatically add a line segment to connect the first point to the last point.

Objects are deleted by picking a "Delete" menu item, and then picking the object to delete. The picked object is deleted and erased as soon as it is selected. "Flipping" objects is performed in the same way. The "Flip" menu item and one of its subitems is selected, then the object to be flipped is picked. There are two "Flip" menu subitems: "Horizontal" and "Vertical." As soon as the object to be flipped is picked, it is erased, flipped around the pick point, and redisplayed.

The individual points in a line or polygon are moved by selecting the "Move" menu item, then selecting a point and dragging it to a new location. The command to move points is an implicit command: either the entire line or polygon, or just one of its points will be moved, based on what part of the object is selected. If the cursor is close to a point (within three pixels) when the select button is pressed, only the point is moved. Otherwise, the entire object is moved.

An object's location and size values can be viewed or edited by picking the "Show/Edit" menu item, then picking the object. A "Show/Edit" requester pops up in the drawing window when the object is picked. The requester contains labels and string gadgets which show the object's current position (minx,miny) and size (width,height). The selected object's values are changed by typing the new values into the labeled string gadgets. Changing the x and y values will move the object to a new position, and changing the width or height values will resize the object. Objects can be flipped by entering negative width or height values. The requester also contains "Modify" and "Cancel" pushbuttons. The requester is popped down by picking either of these pushbuttons. If the "Modify" pushbutton is picked, the selected object's values are updated and it is redisplayed. Otherwise, the object is just unhighlighted.

Each of these new program actions is implemented with event handler procedures and event chaining. Let's briefly review how event handlers and event-specific data are attached to Intuition's input objects (menus, gadgets, etc) and used to process input events. Recall that a pointer to an event handler procedure and a pointer to the data needed by the handler is placed into a structure typedef'ed "intuiExtension_t". A pointer to this structure is then attached to an Intuition input object. If the input object is a gadget or window, the intuiExtension_t structure pointer is placed in the "UserData" member of the gadget or window structure. Since MenuItem structures do not have a UserData member where we can place our intuiExtension_t structure pointer, we define our own menu item structure which includes both Intuition's MenuItem structure and our extension structure. The extended menu item structure is typedef'ed "myMenuItem_t." When an input event is received in "handleInput( )," the intuiExtension_t structure pointer is retrieved from the input object, and the event handler is called using the function pointer. The event handlers either directly perform an action, such as closing a window, or place information in the program's state vector for use by another event handler. The state vector in our program is a structure called "world."

Let's look at the new program commands, the event handlers for each, and the data sent to the handlers. We'll then see how the handlers and data are used to implement the new commands. Table 1 lists the commands, the handlers responsible for the commands, and the data sent to the handlers.

| Table One | New Program Commands | | |
|-----------|-----------|----------|----------|
| Command | Event | Event Handler | Handler Data |
| Insert-Shape | MENUPICK | miSetAction | insShapeData |
| Insert-Line | MENUPICK | miSetAction | insLineData |
| Insert-Polygon | MENUPICK | miSetAction | insPolyData |
| Delete | MENUPICK | miSetAction | deleteData |
| Flip-Horizontal | MENUPICK | miSetAction | flipXData |
| Flip-Vertical | MENUPICK | miSetAction | flipYData |
| Show/Edit | MENUPICK | miSetAction | showEditData |
| Modify | GADGETUP | updateValues | seRequester |
| Cancel | GADGETUP | cancelRequester | seRequester |

The last two commands in Table 1 are the "Show/Edit" requester commands which update an object after its values are modified, or cancel an edit action. The interface objects for these two commands are pushbuttons in the "Show/Edit" requester. The data sent to the event handlers for both of these commands are pointers to the requester containing the pushbuttons. The rest of the commands in Table 1 are menu commands. The data sent to miSetAction( ) for each of the commands are instances of a structure typedef'ed "miData_t." This structure contains a function pointer to an "action" function, an action modifier, and a pointer to whatever data the action function needs to do its job. Table 2 shows the miData_t structure members for each of the new menu commands.

| Table Two | New Menu Command Handler Data Elements | | |
|-----------|-----------|----------|----------|
| Structure Name | Action Procedure | Modifier | Action Data |
| insShapeData | dragAndInsert | None | shapeClass |
| insLineData | dragAndInsert | None | lineClass |
| insPolyData | dragAndInsert | None | polyClass |
| deleteData | pickAndDelete | None | None |
| showEditData | pickAndEdit | None | None |
| flipXData | pickAndFlip | FLIPX | None |
| flipYData | pickAndFlip | FLIPY | None |

The action data for the first three structures are pointers to other structures. We will discuss these three structures in more detail below. They are used in "dragAndInsert( )" to create shape, line, or polygon objects. The modifiers stored with the flip data are used in "pickAndFlip( )" to determine which direction to flip an object.

## Program File Organization

Since our mini-CAD program now has quite a bit of code, the source code has been split up and placed in several files, and the data structures and procedures in Table's 1 and 2 are no longer in a single file. The following is a complete list of all the program files and brief descriptions of their contents.

### Program source code and include files

| | |
|---|---|
| globalDefs.h | global structure definitions and declarations. |
| ieHandlers.h | declaration of the event handlers and action procedures defined in tstObject.c |
| tstObject.c | main program, event handlers, action procedures, and Intuition-related procedures. |
| menu.c | menu definitions and menu procedures. |
| menu.h | declaration of menu array, menu action modifiers, and menu procedure declarations. |
| handleInput.c | defines handleInput( ) and handleDrag( ). |
| handleInput.h | declarations for handleInput( ) and handleDrag( ). |

### Transform source code and include files

| | |
|---|---|
| transform.c | 2d transform and matrix procedures |
| transform.h | declarations and macros for transform and matrix procedures. |

### Shape object source code and include files

| | |
|---|---|
| shape.c | shape class and object interface procedures. |
| shape.h | declarations for shape class interface procedures. |
| shapeClass.c | implementation file for shape class and shape objects. |
| shapeClassP.h | private include file for shape class. |
| shapeClass.h | public include file for shape class. |
| shapeUtil.c | geometric utility procedure definitions.shape |
| Util.h | declarations for geometric procedures. |

### Line object source code and include files

| | |
|---|---|
| line.c | line class and object interface procedures. |
| line.h | declarations for line class interface procedures. |
| lineClass.c | line class and line object implementation file. |
| lineClassP.h | private include file for line class. |
| lineClass.h | public include file for line class. |

### Polygon object source code and include files

| | |
|---|---|
| polyClass.c | polygon class and object implementation file. |
| polyClassP.h | private include file for polygon class. |
| polyClass.h | public include file for polygon class. |

### Make/Link files

| | |
|---|---|
| tstObject.make | makefile for compiling the executable program, tstObject.tst |
| Object.lnk | link file for linking object modules. |

## Implementing the New Program Commands

Now that we know what the event handlers for our new commands are, know where they are located, and know how to interactively use the commands, let's look at how the commands are implemented. All Intuition events are received in handleInput( ). When an event is received, handleInput( ) retrieves the pointer to the intuiExtension_t structure stored in the input object's structure. The function whose pointer is in the intuiExtension_t structure is then called. For all our new menu commands,

miSetAction( ) is the procedure called by handleInput( ). MiSetAction( ) is simple: it places the data sent to it into the program state vector (named "world") and returns. MiSetAction( ) sets the "action", "modifier", and "data" members of the "world" structure equal to the corresponding values in the miData_t structure pointer sent to it. If the next event received by handleInput( ) is a MOUSEBUTTONS event, windowEvent( ) is called to take care of it. For select button events, WindowEvent( ) calculates the world coordinates corresponding to the window coordinates where the button was pressed. It then stores both the world and window coordinates in the state vector. Finally, if the action member of the state vector contains a valid action procedure pointer, the action procedure is called. Our new action procedures are dragAndInsert(), pickAndDelete(), pickAndEdit( ), and pickAndFlip( ).

DragAndInsert( ) is responsible for creating new objects. It retrieves the "class" structure pointer from the state vector and calls createShape( ), which creates new objects by allocating and initializing memory for them. After an object is created, insertDrag( ) is called. InsertDrag( ) takes care of the drag interaction for establishing initial locations and sizes for new objects. The new objects are then added to the list of world objects and displayed. InsertShape( ) adds objects to the list of world objects, and displayShape( ) draws them.

The action procedure for deleting objects is pickAndDelete( ). It calls findObject( ) to see if an object is located at the coordinates where the select button was pressed. If an object is found, it is erased, removed from the list of world objects, and deleted. EraseShape( ) erases any object, removeShape( ) takes care of removing them from the list of objects, and deleteShape( ) frees memory allocated for objects.

The action procedure which pops up the "Show/Edit" requester is pickAndEdit( ). This procedure determines which object was picked, highlights it, and then queries the object to determine its current location and size values. The position and size values are placed in the requester's string gadgets, and the requester is popped up. The pointer to the picked object is saved in the state vector so the action procedures which pop down the requester can tell which object was selected. This is all that pickAndEdit( ) needs to do, so it returns. While the requester is displayed, the values shown in the requester's string gadgets can be edited. The "Show/Edit" requester is popped down when its "Modify" or "Cancel" pushbutton is picked. When the "Modify" pushbutton is selected, its action procedure, updateValues( ), is called by handleInput( ). This action procedure reads the edited values from the requester's string gadgets, pops down the requester, erases the selected object, and updates it with the new values. CancelRequester( ) is the action procedure called when the requester's "Cancel" pushbutton is selected. All it does is pop down the requester and unhighlight the selected object.

An object's data values are obtained by calling getShapeValues( ), and they are updated by calling setShapeValues(). GetShapeValues() is called from pickAndEdit( ), and setShapeValues( ) is called from updateValues( ). Each of these procedures is sent an array of name-value pairs. "Name" is a data element identifier, and "value" is a pointer to a variable containing a value. GetShapeValues( ) returns the object's value corresponding to "name" in the value pointer. SetShapeValues( ) places the pointed-to value into the object's data element identified by "name." Querying and updating objects without knowing anything about them is a tricky problem. We'll take a look at the technique used by getShapeValues() and setShapeValues() to solve this problem a little later.

PickAndFlip() flips objects horizontally or vertically. It first finds the picked object, then examines the action modifier (FLIPX or FLIPY) stored in the state vector to determine which way to flip the object. If the object is being flipped horizontally, the (x,y) scale values are set to -1.0 and 1.0, and if it is being flipped vertically, the (x,y) scale values are set to 1.0 and -1.0. PickAndFlip( ) then erases the object and calls resizeShape( ), sending it the "flip" reference point (pick point) coordinates and the flip scale values. ResizeShape( ) takes care of scaling objects. After resizeShape( ) returns, the selected object is redisplayed, and pickAndFlip( ) is done.

Moving the individual points in a line or polygon is a common CAD operation. Points are moved the same way entire objects are moved: the point to be moved is selected, then dragged to a new location. DragAndMove( ), the move action procedure we developed last time, has been enhanced to move points belonging to a line or polygon. After dragAndMove( ) finds a selected object, it calls findPoint( ). If the selected object does not contain points which can be moved (shapes), findPoint( ) returns a NULL pointer. However, if the selected object is a line or polygon, the pick coordinates are used to determine if the pick point is within a tolerance distance (3 pixels) of one of the line's or polygon's points. If so, a pointer to the point is returned. If findPoint( ) returns a non-NULL pointer, dragAndMove( ) calls dragPoint( ) and movePoint( ) to move the point. Otherwise, moveDrag( ) and moveShape( ) are called to move the selected object.

Take another look at dragAndInsert( ). It creates shapes, lines, and polygons without knowing what kinds of objects it is creating! If you look at the rest of the event handlers and action procedures in our program, you will see that none of them know what kinds of objects they are managing! DragAndInsert( ), the event handlers, and the action procedures in our program illustrate two important features of object-oriented programming: data hiding and polymorphism. The only information our program knows about the objects in it is how to call their procedures. It does not even know the actual types of the objects. To our application code, all objects are of type "object_p", which is an anonymous type used to hide an object's real type. Notice also that all objects, whether they are shapes, lines, or polygons, are displayed by calling displayShape( ). Internally, displayShape( ) uses a function pointer to call the correct object drawing procedures without knowing what procedure it is calling. If the object is a shape, the code to draw shapes is called, and if it is a polygon, the polygon drawing procedure is called. "Polymorphism" (many forms) is the term used to describe this feature of object-oriented programming. In the remainder of this article, we will take a closer look at the concepts and characteristics of object-oriented programming, and develop an object-oriented "framework" for implementing geometric objects. Let's get started with "objects" and "classes."

## Objects, Classes and Abstract Data Types

The "object" and "class" concepts in object-oriented programming have the same meaning as they do in everyday language. Things (objects) which are similar are grouped into cat-

egories called classes. Similar classes, in turn, are grouped into categories of superclasses and subclasses. Mammals, for example, are a subclass of a class of living creatures called animals. Canines are a subclass of mammals. Subclass objects are similar to their superclass objects, but are also different in some recognizable way. Software objects are grouped into software classes the same way that real objects are grouped into classes: all software objects which share the same data definition and the same functionality belong to the same software class. The individual software objects which belong to a class are called "instances of the class," or "class instances."

Software classes are more than just conceptual groupings of similar objects. They are programmer-defined data types. They consist of definitions of their instance objects' data structures, code implementing their objects' behavior, and a mechanism for linking instance objects to their code. In C++, the "class" construct is a feature of the language. It provides support for programmer-defined data types, which are managed automatically in almost the same way built-in data types such as integers and doubles are managed. Since we are not using C++, we have to create our classes and manage them ourselves. The basic programming construct for creating programmer-defined data types in C is "data abstraction."

An abstract data type consists of a data definition, the code for managing and manipulating the type (data definition), and a description of how to use the type. The transform structure and code we created in Part 1 is an example of an abstract data type. We created a data structure, typedef'ed it as "transform2_t," and wrote all the code needed for using transform2_t structures. We created a programmer-defined data type. To use transform2_t's, we do not need to know anything about their internal data structure or how their code works. All we need to know is which procedures we can use with them, and how to call the procedures. We implement and package any abstract data type the same way we did our transform2_t data type: all the code for the data type is placed into a single source code (xxx.c) file, and the structure and procedure declarations programs need to use the data type are placed in an include (xxx.h) file. This is the way we packaged our transform data type. The structure definitions and code for object-oriented objects are packaged in a similar way. Classes consist of the "package" of code, structure definitions, and structure declarations for a single type of object. A class is the implementation of an abstract data type, and objects which belong to the class are "variables" of the data type.

## Data Hiding

The only problem with abstract data types is that the implementation details (data structures and code) for the types are usually visible to programs using them. Since the implementation details are visible, the data elements for abstract data types can be directly accessed and modified. This is a temptation most programmers easily succumb to, usually under the banner of "efficiency." To remove this temptation, the data structures and code for object-oriented objects are hidden from applications which use them.

To hide class implementation details, the prgram places the structure and procedure declarations for a class's objects into two include files. One include file is "public" and contains declarations needed by applications. The other include file is "private"

and contains declarations needed by the class. As an example, the three files containing the implementation of shapeClass are shapeClass.c, the source code file, shapeClassP.h, the private include file, and shapeClass.h, the public include file. For our object-oriented implementation, brand new classes need two additional files. One is a source code file containing application-class "interface" code, and the other is an include file for programs using the interface code. Shape.c and shape.h are the two interface files for shapeClass. All the include files for our three object classes (shapeClass, lineClass, and polyClass) are in Listings 1 through 8.

In our mini-CAD program, all objects are visible only as an opaque pointer type, "object_p" (actually a "void *"). The real structure definitions are contained in the private include files. Unless the private files are included in our program (cheating!), we cannot access or modify the data elements for our objects. Further, all the internal procedures for each of our object types are declared "static.". Since static procedures are not visible outside their file, we can not directly call the procedures which implement the functionality for our objects. This brings up a question: If both the data and the code for objects are hidden, how can the objects even be used?

## Binding Data to Code with Pointers

I mentioned above that classes contain a mechanism for linking the data for their objects to their code. This mechanism is a structure containing function pointers which point to the code for the class's objects. The structure containing the function pointers is called a "class structure." It is declared in the private include file for a class and initialized in the class's source code file. A pointer to the class structure is made available to code using the class's objects as an opaque type in the class's public include file. The opaque class structure pointers in our program are named "shapeClass," "lineClass," and "polyClass." Their opaque type is "class_p.". The class structure pointers are used by our program when it calls createShape( ) to create objects.

The first structure member of our object structure is a pointer to its class structure. The class structure pointer member of an object is initialized when the object is created. By including the class structure pointer as part of an object's data, objects are indirectly linked to their code. Figure 2 shows how object structures, class structures, and object procedures are all linked together with structure and function pointers. Since the data structure for an object contains a pointer to its class structure, which in turn contains pointers to the class's procedures, an object's procedures can be indirectly called using its data structure. Here is a code fragment showing how the "display" procedure for shape objects is called:

```
shapeObject_p object; /* shape object struct pointer */
shapeClass_p  class;  /* shape class struct pointer  */

class = object->class;        /* get class pointer    */
(*class->display)(object,window); /* call display     */
```

Because an object's procedures (operations) can be called through the class pointer which is part of the object, object-oriented objects are "active" data, and are sometimes referred to as "actors."

It is apparent from the above code fragment that to correctly manage all the structure and function pointers associated with objects, knowledge of at least part of the implementation details of objects and classes is needed somewhere. This knowledge consists of several pieces of information. Some code needs to know that object structures contain class pointers and that class structures contain function pointers. In addition, knowledge of which function pointers are contained in a particular class structure, and what their linkage variables are, is also needed. We do not want this kind of knowledge embedded in our CAD application code, since it would have to perform type checking and type casting to correctly use our objects. In fact, as mentioned above, this kind of knowledge is not even available to our application code.

When a new class is created, "interface" code for using the class and its objects is also created. The interface code is a set of procedures which provide access to a class's objects and their functionality. The interface code for a class is referred to as its "message interface" by object-oriented programmers, and calling one of the interface procedures is referred to as "sending a message" to an object. Most class interface procedures do no more than the code fragment shown above: they just retrieve the class structure pointer from an object, then retrieve a function pointer from the class structure pointer and call the procedure.

The interface procedures for our objects are in the files shape.c and line.c. Shape.c contains interface procedures for using any shape object, and line.c contains procedures for using line and polygon objects. The code in shape.c knows that shape objects contain pointers to a shapeClass structure, knows which function pointers are in the shapeClass structure, and knows how to call the shape object procedures through the function pointers. The code in line.c contains the same information about line object structures and the lineClass structure. The procedures in shape.c and line.c are the ones called by the "action" procedures which implement all of our CAD functionality.

The interface procedures for a class do more than simply provide access to the class's objects. One important function they perform is shielding application-specific program code from changes in data structures and code. The way classes and objects are defined and implemented can be changed without affecting the way they are used and accessed in application programs, as long as the way the interface procedures are called does not change. In addition, when combined with structure overloading, the interface procedures provide an important feature of object-oriented programming known as "polymorphism."

## Structure Overloading, Subclasses, and Polymorphism

If we analyze all the different types of operations needed in CAD programs, we see that most CAD operations are common to all objects, whether they are circles, lines, text strings, or others. All CAD objects need display, erase, highlight, rotate, and scale operations, among others. However, some object types also need to perform operations which are not performed by other objects. Lines and polygons need operations for manipulating their individual points, such as an operation for moving the points. Circles and text objects do not need these operations. In addition to a common set of operations, graphical objects also have a considerable number of common data elements. In our mini-CAD program, the data structures for all our objects contain pointers to

their class structures, linked list "next" pointers, and bounding box coordinates. If we group the members of our structures so that those members which are common to different objects are in the same positions in their structures, we can use "structure overloading" with our classes and objects. Structure overloading is a technique for defining new structures by adding structure members to the end of existing structures. The technique allows code written for existing structures to be used for the new structures. This technique is extensively used in object-oriented programming, and is also widely used in the Amiga system software (message and library structures, for example).

For our graphical objects and classes, the data elements common to all graphical objects are placed in the structure for shape objects, and the function pointers for functions common to all graphical objects are placed in the class structure for shapeClass. These structures are defined in shapeClassP.h (Listing 1). Notice that macros are used to define the structure members. Macros are used because they reduce the effort required to overload the structures and change structure definitions. Now take a look at the class and object structure definitions for lineClass and lineObject in lineClassP.h (Listing 2). The structure definition for lineClass is almost identical to the structure definition for shapeClass. The only difference is that lineClass extends (overloads) shapeClass, adding three new function pointers. Similarly, the structure definition for lineObject extends shapeObject's structure by adding a "points" pointer to shapeObject. Finally, take a look at the structure definitions for polyClass and polyObject in polyClassP.h (Listing 3). Other than containing empty macro definitions as placeholders, these structures are identical to the lineClass and lineObject structures. These are all examples of overloaded structures. Let's line up our structure definitions side by side and compare their members. Table 3 compares our class structures and Table 4 compares our object structures.

| Table Three | Class Structure Members | |
|---|---|---|
| shapeClass_t | lineClass_t | polyClass_t |
| CLASS_PART | CLASS_PART | CLASS_PART |
| SHAPEC_PART | SHAPEC_PART | SHAPEC_PART |
| .................... | LINEC_PART | LINEC_PART |
| .................... | .................... | POLYC_PART |

| Table Four | Object Structure Members | |
|---|---|---|
| shapeObject_t | lineObject_t | polyObject_t |
| OBJECT_PART | OBJECT_PART | OBJECT_PART |
| SHAPE_PART | SHAPE_PART | SHAPE_PART |
| .................... | LINE_PART | LINE_PART |
| .................... | .................... | POLY_PART |

Structure overloading can significantly reduce code volume and eliminate code duplication. We can see why this is true from our class and object definitions. Since the definitions of lineClass and polyClass include the definition of shapeClass, all the code which "knows" about shapeClass can be used to manage

and manipulate the shapeClass part of the lineClass and polyClass structures. In the same way, the code which knows about lineClass can be used for the lineClass part of the polyClass structure. These statements are also true for our overloaded object structures. In our mini-CAD program, shape.c contains the code which is used for all our overloaded class and object structures, and line.c contains the code for the overloaded line and polygon classes and objects. As far as the procedures in shape.c are concerned, all class structures are shapeClass_t's and all object structures are shapeObject_t's. The procedures in line.c view all the class and object structures sent to them as lineClass_t's and lineObject_t's.

Let's return to our "display" example and see what happens when objects are displayed using function pointers from our overloaded structures. Suppose we have created and added a shape object and a line object to our model world, and are ready to redisplay all (both!) of our objects. To do this, we call "drawAll( )", our application procedure which redraws all our objects. Here is the code from drawAll( ) which redisplays the objects:

```
obj = world.objects;

while( obj )
{
    (void)displayShape(obj,window);
    obj = nextShape(obj);
}
```

DisplayShape( ) and nextShape( ) are defined in shape.c, and here are their definitions:

```
int displayShape( object_p object,window_p window )
{
    class_p      class = getObjectClass(object);
    shapeClass_p shapeC = getShapeClass(class);

    if ( ! object || ! shapeC || ! shapeC->display )
        return 1;
    return (*(shapeC->display))(object,window);
}
...

object_p nextShape( object_p object )
{
    shapeObject_p  shape = (shapeObject_p)object;

    if ( ! object )
        return NULL;
    return (object_p)shape->next;
}
```

The first thing displayShape( ) does is retrieve the class pointer from whatever object is sent to it (getObjectClass and getShapeClass are macros defined in shapeClassP.h to retrieve and cast class pointers). If the object is a shape object, the class pointer points to shapeClass. If it is a line object, the class pointer points to lineClass, but displayShape( ) does not know this. Since the lineClass structure overloads the shapeClass structure, it can be safely cast to the type, "shapeClass_p", and its members can be accessed and used as if it really were a shapeClass_p structure pointer. After making sure none of the pointers are NULL, displayShape( ) uses the display function pointer to call the

object's drawing procedure. For shape objects, the procedure which draws shapes as rectangles is called, and for line objects, the procedure which draws polylines and polygons is called.

NextShape( ) works in a similar manner. It casts whatever object pointer is sent to it to a shapeObject pointer, and simply returns the 'next' structure member stored in every shapeObject structure. Since lineObject and polyObject structures overload shapeObject structures, nextShape( ) can access these structures in exactly the same way it accesses shapeObject structures.

As these two examples show, the procedures which are used for shape objects can also be used for line objects and poly objects, since the first part of their structure definitions are identical to the structure definition for shape objects. This means that either line objects or poly objects can be used wherever shape objects can be used. However, the opposite is not true; that is, shape objects can not be used wherever line objects are used, nor can line objects be used wherever poly objects are used. Imagine what would happen if the procedures in line.c cast the shape class structure pointer to a line class structure pointer and tried to call one of the procedures line class adds to shape class! Most likely, the software gremlin which lives in all code would probably wake up and go to work! The generic interface procedures written for an extended object structure prevent this from happening by looking at an object's class structure to determine whether it contains the necessary extensions before attempting to access them. This is similar to type checking. The procedure isClassInstance( ) in shape.c compares an object's class structure pointer with an input structure pointer to see if they match. Here is a code fragment showing how the generic procedure findPoint( ) in line.c calls isClassInstance( ) to verify the object sent to it is a line object or an object which extends line objects before it calls the findPoint procedure line class adds to the shape class structure definition:

```
/*
 * make sure the object is either a lineObject or
 * an object which is an instance of one of line
 * class's subclasses.
 */
if ( ! isClassInstance(object,lineClass) )
    return NULL;

if ( lineC->findPoint )
    return (*(lineC->findPoint))(object,px,py,eps);
```

And here is the definition of isClassInstance( ):

```
int isClassInstance( object_p object,class_p class )
{
    class_p       objClass = getObjectClass(object);
    shapeClass_p  shapeC   = getShapeClass(objClass);

    while ( shapeC )
    {
        if ( (class_p)shapeC == class )
                return 1;
        shapeC = getShapeClass(shapeC->superClass);
    }
    return 0;
}
```

As we will see shortly, each class structure contains a pointer to the structure it overloads. The pointer is the structure member named "superClass". isClassInstance( ) first retrieves the class structure pointer from the input object. Inside the "while" loop, the retrieved pointer is compared with the input class pointer, and if they match, isClassInstance( ) returns 1 to indicate the class pointers match. Otherwise, the pointers to the overloaded class structures are retrieved from each class structure and compared with the input class pointer. The loop continues until the root class structure (shapeClass) has been reached and compared. When the code in line.c calls isClassInstance( ) and sends it a lineObject, the comparison succeeds immediately. If the object is a polyObject, the first comparison fails, since polyClass does not match lineClass. The second comparison succeeds, since the superClass pointer retrieved from the poly class structure points to the line class structure. However, if the object is a shape object, the comparison will fail, since the shape class structure does not overload the line class structure.

When our application code calls the interface procedures in shape.c and line.c, it does not know what kind of object it is sending to the procedures, and the called procedures do not know the "real" type of object sent to them. Nevertheless, the correct procedures for our different classes of objects are called. In object-oriented terminology, this type of functionality is called polymorphism, and it is one of object-oriented programming's more powerful features. Polymorphism allows us to remove type checking from our code—no more massive switch statements to decide which procedures to call, and no more special case code for different objects—we just call a procedure and the "right actions" are performed! What is almost magical about polymorphism is that we can add brand new objects and classes—created by overloading the shape class and shape object structures—to our program, and our existing code will automatically work without any changes or additions! Think about the significance of these statements.

From a conceptual viewpoint, lineClass and lineObject are specialized versions of shapeClass and shapeObject, and polyClass and polyObject are specialized versions of lineClass and lineObject. Our overloaded structures form a hierarchy of class structures, together with their instance objects' structures. ShapeClass is a "superclass" for both lineClass and polyClass, and lineClass is a superclass for polyClass. Another way to describe the same hierarchical class relationships is by saying polyClass is a "subclass" of both lineClass and shapeClass, and lineClass is a subclass of shapeClass. Subclasses specialize existing classes by adding new data or functionality, by modifying existing functionality, or both. With a few pointers and a couple of programming techniques, we can use structure overloading to implement "true" subclasses and "method inheritance," another powerful feature of object-oriented programming.

## Inheritance

It was mentioned above that all the procedures in a class are defined as static, and thus are not visible outside their definition file. This would seem to be a major flaw in object systems, since static procedures cannot be shared among different classes, even subclasses of a class which defines a static procedure. However, object systems use a technique called "method inheritance" to

remove this restriction. Method inheritance enables subclasses to use procedures defined by their superclasses without knowing the name of the procedures or where they are defined.

You may have noticed in our CAD program that all objects are moved, resized, and rotated in exactly the same way. There is a good reason for this: all our classes use the same drag procedures for moving, sizing, and rotating objects. The procedures are moveDrag( ), sizeDrag( ), and rotateDrag( ), and they are static procedures defined in shapeClass.c. LineClass and polyClass inherit these procedures from shapeClass.

To implement method inheritance, we either need a way for subclasses to get function pointers from their superclasses, or we need a way for superclasses to initialize function pointers for their subclasses. We will implement method inheritance the second of these two ways. Four requirements must be fulfilled for this to work. They are:

1. Each class which implements an inheritable method defines a special "inherit" symbol for the method, and the special symbol is made visible to subclasses. Subclasses which need to inherit a method place the special symbol in the function pointer member corresponding to the procedure to be inherited.
2. Each class needs a setup( ) method which initializes the structure members for its class structure and the class structures for its subclasses.
3. Each class needs to keep a pointer to the class structure of its superclass so it can call the setup( ) method of its superclass.
4. All method inheritance for a class and its superclasses must be resolved before any class methods other than setup( ) can be called. This requirement is met by calling a class's setup( ) method the first time one of its objects is created.

Let's look at the polyClass structure to see how it is initialized, then step through code fragments from all three of our classes to see how polyClass inherits moveDrag( ) from shapeClass and inherits display( ) from lineClass. Notice that almost all of polyClass's function pointer members are named "inherit... ." These are the names of the "special symbols" defined by shapeClass and lineClass so the methods they define can be inherited. They are the names of procedures which do nothing except return an error code. The "inherit..." procedures are declared in the private include file for a class, and defined in the class's source code file. Since subclasses include the private include file of their superclasses, the names of the dummy "inherit" procedures are visible to them. Subclasses let their superclasses know which methods they want to inherit by placing pointers to the dummy procedures in their class structure. This is the class structure definition for polyClass:

```
static shapeClass_p superClass =
                        (shapeClass_p)&_lineClass;
polyClass_t _polyClass =
{
(class_p)&_lineClass,    /* superclass      */
0L,                      /* flagWord        */
sizeof(polyObject_t),    /* objectSize      */
/* methods specified by shapeClass          */
setup,
NULL,                    /* initialize      */
```

```
NULL,                    /* deallocate            */
setvalue,
getvalue,
update,
inheritDisplay,
inheritErase,
inheritHighlight,
inheritUnhighlight,
insertdrag,
inheritMovedrag,
inheritSizedrag,
inheritRotatedrag,
inheritMove,
inheritResize,
inheritRotate,
inheritPointToObject,
inheritExtent,
/* methods specified by lineClass            */
inheritFindPoint,
inheritDragPoint,
movePoint,
};

const class_p polyClass = (class_p)&_polyClass;
```

Method inheritance is triggered when createShape( ) is called. For polygons, the class structure pointer sent to createShape is polyClass. CreateShape( ) casts the class pointer to "shapeClass_p" and looks at the flagWord member to see if the class has been initialized. If not, the setupClass( ) procedure for the class is called using the function pointer to the class's setup method. Here is the code which does this:

```
object_p createShape( class_p class )
{
    shapeClass_p   shapeC = getShapeClass(class);
    shapeObject_p  shape = NULL;

    /* make sure class is initialized            */

    if ( ! (shapeC->flagWord & IS_INITIALIZED) )
        if ( *(shapeC->setupClass)( class ) )
                return NULL;
    ...
}
```

Since the class pointer sent to createShape points to polyClass, createShape( ) calls polyClass's setup method. PolyClass keeps a pointer to its superclass (lineClass), and after it completes its setup processing, calls its superclass's setup method, sending its class pointer to it. Here is the code from polyClass's setup( ) method:

```
static int setup( class_p class )
{
    polyClass_p polyC = (polyClass_p)class;
    ...
    (void)(*(superClass->setupClass))(class);
    polyC->flagWord |= IS_INITIALIZED;
    return 0;
}
```

LineClass defines a display method which draws polylines and polygons. This is the procedure polyClass wants to inherit. LineClass's setup method examines the "display" function pointer member of the class pointer sent to it. If the display function pointer points to inheritDisplay( ), it is replaced with the pointer to the display( ) procedure defined by lineClass. After lineClass's setup procedure finishes its processing, it calls shapeClass's setup procedure:

```
static int setup( class_p class )
{
    lineClass_p lineC = (lineClass_p)class;
    ...
    if (lineC->display == inheritDisplay )
    lineC->display = display;
    ...
    (void)(*(superClass->setupClass))(class);
    lineC->flagWord |= IS_INITIALIZED;
    return 0;
}
```

ShapeClass also defines a display method which can be inherited, so it also looks for a function pointer to inheritDisplay( ). However, since the inheritDisplay( ) function pointer has already been replaced by a pointer to lineClass's display method, shapeClass does not find the inheritDisplay( ) pointer in the polyClass structure. ShapeClass's setup method looks for function pointers to all the inherit...( ) procedures defined by shapeClass, so it finds the pointer to inheritMovedrag( ) in the polyClass structure. It then replaces the inheritMoveDrag( ) function pointer with the pointer to the movedrag( ) procedure defined for shape objects:

```
static int setup( class_p class )
{
    shapeClass_p shapeC = (shapeClass_p)class;
    ...
    if (shapeC->display == inheritDisplay )
      shapeC->display = display;
    ...
    if (shapeC->movedrag == inheritMovedrag )
      shapeC->movedrag = movedrag;
    ...
    shapeC->flagWord |= IS_INITIALIZED;
    return 0;
}
```

When shapeClass's setup( ) procedure finishes, all the inherit...( ) function pointers in the polyClass structure have been replaced by procedures defined by either lineClass or shapeClass. The way method inheritance works and can be implemented should be clear from this example. Its significance should also be apparent: very little code is required to implement fully-functioning polygon objects, since almost all of their methods are inherited.

You are probably wondering why such an elaborate scheme is used to implement inheritance. Why not just make class procedures visible to their subclasses, and let the subclasses explicitly initialize their own function pointers? The main reason we have implemented inheritance as illustrated is that subclasses can automatically inherit procedures without knowing which of their superclasses defines the inherited procedures. This technique

enables us to add new procedures to a class without having to change its subclasss's class structures. For example, suppose we define a moveDrag( ) procedure for lineClass which will draw polylines as they are being dragged around, instead of using shapeClass's moveDrag( ) procedure which draws a box. The inheritance scheme we have implemented will result in polyClass, and also any future subclasses of lineClass, automatically inheriting the new moveDrag( ) procedure. Automatic inheritance is one of the reasons object-oriented systems are ideal for rapid prototyping: new object classes can be implemented quickly with very little new code. Later, when the new classes are working correctly, methods which are more appropriate for the new objects can be written to replace the methods which were inherited during prototyping.

Object systems employ several other techniques which enable subclasses to use methods defined in their superclasses. One technique is "method chaining," and another technique is "superclass dispatching." Superclass dispatching occurs when a method in a subclass calls a corresponding method in its superclass. The example just discussed uses superclass dispatching: the setup( ) procedure in each class calls the setup( ) procedure of its superclass. Method chaining is more complex, and is best explained with an example. Method chaining is generally used when objects are created and destroyed.

## Creating and Destroying Objects

All our graphical objects are dynamically created at runtime in dragAndInsert( ), which calls createShape( ) to actually create an object and return its pointer. Several steps have to occur when objects are created: first, memory for the object needs to be allocated; next, the object has to be "bound" to its class structure; finally, the object has to be initialized. Allocating memory for an object and binding it to its class structure is easy. Initializing an object is a bit tricky. Here is the object creation code from createShape( ) and from the static procedure initialize( ) in shape.c which performs these steps:

```
object_p createShape( class_p class )
{
    shapeClass_p    shapeC = getShapeClass(class);
    shapeObject_p   shape = NULL;
    ...
    /*  allocate memory for the object           */
    shape = (shapeObject_p)malloc(shapeC->objectSize);
    ...
    shape->class = class;
    shape->next = NULL;
    /*  initialize the object                     */
    if ( initialize( (object_p)shape,class ) )
    {
       free( (void*)shape );
       return NULL;
    }
    return (object_p)shape;
}

static int initialize( object_p object,class_p class )
{
    class_p super = getSuperClass(class);
    shapeClass_p  shapeC = getShapeClass(class);
    /*
     * recursively call self until we get to the root
```

```
 *  class (has no superclass)
 */

    if ( super )
       if ( initialize(object,super) )
               return 1;
    /*  call each class as we return from recursion    */

    if ( shapeC->initialize )
       return (*(shapeC->initialize))(object);
    return 0;
}
```

The only variable sent to createShape( ) is a pointer to the class structure for the object being created. As noted earlier, the class structure pointer is the menu "action data" for the "insert" menu item. CreateShape( ) casts the class structure pointer to a shapeClass pointer, allocates memory using the object size information stored in the class structure, and binds the new object to its class structure pointer. CreateShape( ) then calls initialize( ) to take care of initializing the rest of the object's data values. The new object is initialized using method chaining: the initialize( ) methods for the object's class and all its superclasses are called, from the "root class" (shapeClass) first down to the object's class. This form of method chaining is called "downward chaining," and it is used to permit every superclass to initialize the object structure members they are responsible for. The object's superclasses are called in superclass to subclass order so that each subclass can "override" (change) any values initialized by their superclasses. If a subclass does not need to perform any initialization other than that done by its superclasses, it does not need to provide an initialize( ) method. Initialize( ) uses recursion to follow superclass pointers until it reaches the root class, shapeClass. As the recursive calls "unwind," initialize( ) calls the initialize method for each class which has one.

Objects are destroyed in a similar manner, except "upward chaining" instead of downward chaining is used. When upward chaining is used, class methods are called in subclass to superclass order. Here is the object destruction procedure from shape.c showing how upward chaining is used when objects are destroyed:

```
int deleteShape( object_p *object )
{
    class_p         class = getObjectClass(*object);
    shapeClass_p    shapeC = getShapeClass(class);
    ...
    /*
     *  call superclasses in subclass to superclass order
     *  to perform postdestruction object cleanup.
     */

    while ( shapeC )
    {
       if ( shapeC->deallocate )
               (void)(*(shapeC->deallocate))(*object);
       shapeC = getShapeClass(shapeC->superClass);
    }
    /*  free the object                           */

    if ( *object )
       free( (void*)*object );
    *object = NULL;
    return 0;
}
```

Upward chaining occurs inside the "while" loop in deleteShape( ). The code looks at the deallocate function pointer in the class structure pointer, and if it is not NULL, calls it. Next, the pointer to the class's superclass structure pointer is retrieved, and if the superclass has a deallocate method, it is called. The while loop ends after the root class (shapeClass) is reached. DeleteShape( ) then frees the memory allocated for the object structure.

Why are objects destroyed this way? Why not just free the memory allocated for an object? Any object may contain pointers to internally-allocated memory, and if so, the internal memory needs to be freed before the memory allocated for the object structure is freed. DeleteShape( ) does not know what kind of object is being destroyed, and does not know if it contains pointers to dynamically-allocated memory. Chaining through all the deallocate methods for a class and its superclasses insures that any class which allocates memory for its objects can also free the memory before the object memory is freed.

Object creation, initialization, and destruction are complex operations in object-oriented programming systems. Many details have to be taken care of to insure these operations are performed correctly and completely. Two other complex operations are generically accessing and modifying an object's data values.

### Accessing and Updating Object Data Values

PickAndEdit( ) calls getShapeValues( ) to determine data values for an object so they can be displayed in our program's "Show/Edit" requester, and updateValues( ) calls setShapeValues( ) to update an object's data values after they are modified. Both getShapeValues( ) and setShapeValues( ) use a technique which is common in object systems: an array of structures containing "tags" and values is filled in and sent to a method which queries or modifies an object's values. The "tags" identify data elements which are to be returned or updated, and the "values" are either actual values or pointers to variables which hold the actual values. Tags are identifiers which uniquely identify an object's data elements. They are defined in many different ways by different object systems: some use actual strings, some use macro definitions, and still others use hash functions to generate tags from data element names. Our program uses a simple technique for generating tags. Each object structure member is assigned a unique name in the class responsible for the structure member, and a pointer to the string which points to the name is defined. The pointer to the string pointer is exported as an "atom" in the class's public include file, and is used as the data element tag. There are several advantages to this approach: it is simple; the compiler and linker will guarantee the tags are unique within a task; pointers can be directly compared for equality; and finally, the name can be accessed if needed.

To make using tag-pointer value pairs easy, we define a structure to hold the pairs, and a macro for setting up the array values. The definitions from globalDefs.h are:

```
typedef char **atom_p;    /* the tag is an "atom"    */

typedef struct _arg
{
    atom_p    argAtom; /* structure member identifier  */
    int       hasVal;  /* 1 if value is being returned */
    void      *argPtr; /* pointer to value variable    */
} arg_t,*arg_p;

#define setArg(arg,n,atom,ptr) \
((arg)[(n)].argAtom=(atom),\
 (arg)[(n)].argPtr=(void*)(ptr))
```

When we need to query or modify an object's data values, we set up an array of arg_t's, initialize the values, and call getShapeValues( ) or setShapeValues( ). Here is a code fragment from pickAndEdit( ) which shows how the arg_t structure array and pointer variables are declared, how the array is initialized, and how getShapeValues( ) is called:

```
arg_t     args[10];
double  px,py,wd,ht;
char    clName[MAX_CLASS_NAME_LEN];
long    n;

/*
 * set up the args array, then call getShapeValues.
 * getShapeValues will place the requested values
 * in the pointers corresponding to the 'atoms'.
 */

n = 0;
setArg(args,n,classNameAtom,clName);    n++;
setArg(args,n,xLocationAtom,&px);       n++;
setArg(args,n,yLocationAtom,&py);       n++;
(void)getShapeValues(obj,args,n);
```

GetShapeValues( ) retrieves the object's 'getValues' method from its class structure and calls it to process the arg_t array. Here is the code from shape class's getValues procedure which returns the class name for shape objects:

```
if ( arg[i].argAtom == classNameAtom )
{
    strcpy((char*)arg[i].argPtr,"shape");
    continue;
}
```

There are two weaknesses in using tag-pointer pairs to access and update data values for objects: first, the tags which a class of objects recognizes must be known; and second, the value pointers must be the correct data type and data size. For example, to access an object's class name, our mini-CAD application needs to know that all objects recognize "classNameAtom" as a tag, and that the type of value associated with the tag is a character array which can hold a maximum of 32 characters. The first weakness is minor, since a class which does not recognize an atom just prints an error message and ignores it. The second weakness can have serious side effects (memory corruption) if a value is assigned to a data type which has too few bytes to hold the value. The tags for a class and its subclasses are made known to appli-

cation code by being declared in the class's public include file. The associated data types for the tag value pointers are placed in comments next to the tags, and applications querying or updating object data values are responsible for making sure the pointed-to variables are the correct type and size. Here are how the declarations from shapeClass.h for classNameAtom and xLocationAtom look:

```
extern const atom_p  classNameAtom; /* char*,32 bytes */
extern const atom_p  xLocationAtom; /* double        */
```

Finally, here is how classNameAtom is defined in shapeClass.c:

```
static char *const _className     = "className";
const atom_p classNameAtom = (atom_p)&_className;
```

Modifying an object's values is more complicated than accessing its values. The values are modified in basically the same way they are accessed. However, after the values are modified, the object needs to know which ones were modified so it can perform any actions required because of the modifications. As an example, when the width and height values for objects are changed to negative values, they need to be scaled (to flip them) and moved back to their original locations. Additionally, subclass objects may need to disallow some modifications which their superclass objects allow. Because of these considerations, setting an object's values using tag-value pointer pairs is a two-step process. Modifying the object's values occurs first. After this is complete, an "update" method for the object is called and sent the modified object, along with an unmodified copy of it. The object's update method then compares its new values with its old values to determine which values changed. The update method can disallow changes by setting the modified values back to their original values, allow some values to change, or allow all values to change. Based on which values changed, the update method can perform any additional processing needed to insure the integrity of the object is maintained. The following code from setShapeValues( ) shows how this two-step process is done

```
/*
 * create a shallow copy of the unmodified object
 * The copy is used by the class to determine which
 * object values were changed.
 */

copy = (object_p)malloc(shapeC->objectSize);
...
memcpy((void*)copy,(void*)object,shapeC->objectSize);

/* set the object values                     */

(void)(*(shapeC->setvalue))(object,args,n);

/*
 * send the updated object and the unmodified copy
 * to the class so it can validate the modifications
 * and perform any actions needed to maintain
 * the integrity of the object
 */

ret = (*(shapeC->update))(object,copy,window);
free( (void*)copy );
return ret;
```

You can look at the setvalues and update methods in both shapeClass.c and lineClass.c to see how objects are modified and see the types of actions required when an object's values are changed.

Other than the procedures for creating and destroying objects, and for accessing and updating an object's data values, the rest of the procedures in shape.c, line.c and the three class modules are relatively easy to understand. In fact, the procedures in the class modules are basically the same ones presented and discussed in Part II of this series.

### Summary

In Part 1 of this series of articles, I promised that we would develop a basic CAD program. We have almost achieved that goal: we developed a library of transform procedures and a set of procedures and techniques for handling Intuition's input events in Part 1 and Part 2. In this article, we developed a framework for implementing geometric models using object-oriented programming techniques, and saw how to implement several classes of geometric objects. In my next article, we will see how new classes of objects are implemented using our object-oriented framework, and add rectangles, circles, and ellipses to our mini-CAD application. We will then have a mini-CAD program which can be easily extended to include both new objects and new functionality.

---

**Listing One**      shapeClassP.h

```
/*============================================================*/
/*
 *  Listing 1.  Private include file for shapeClass.
 *
 *  shapeClassP.h - defines class structure and object
 *                  structure for shapes.
 *
 *  The structures defined here are common to all
 *  geometric classes and objects.  New class and object
 *  structures are created by overloading (extending) the
 *  shapeClass and shapeObject structures defined here.
 *
 *  (c)  Copyright 1991, Forest W. Arnold
 *  All rights reserved.
 */
/*============================================================*/

#ifndef SHAPECLASSP_DEFS
#define SHAPECLASSP_DEFS

#include "globalDefs.h"
#include "shapeClass.h"

/*
 * class flag values
 */

#define IS_INITIALIZED  1

/*
 *  Shape Class Part - defines the shape class part
 *  of a shapeClass structure.  The CLASS_PART
 *  consists of information about the class and its
 *  objects, and a 'setup' procedure which is called
 *  to complete initializing the class structure
 *  when the first object is created.
 *  The SHAPEC_PART consists of function pointers
 *  to the methods for a class.
 */
```

```c
#define CLASS_PART \
  class_p         superClass;\
  unsigned long   flagWord;\
  unsigned int    objectSize;\
  int (*setupClass)(class_p);

#define SHAPEC_PART \
  int (*initialize)(object_p);\
  int (*deallocate)(object_p);\
  int (*setvalue)(object_p,arg_p,int);\
  int (*getvalue)(object_p,arg_p,int);\
  int (*update)(object_p,object_p,window_p);\
  int (*display)(object_p,window_p );\
  int (*erase)(object_p ,window_p );\
  int (*highlight)(object_p,window_p );\
  int (*unhighlight)(object_p,window_p );\
  int (*insertdrag)(object_p,window_p );\
  int (*movedrag)(  object_p,window_p,\
                    double *,double * );\
  int (*sizedrag)(  object_p,window_p,double,double,\
                    double *,double * );\
  int (*rotatedrag)(object_p,window_p,double,double,\
                    double *);\
  int (*move)(object_p,double,double);\
  int (*resize)(object_p,double,double,double,double);\
  int (*rotate)(object_p,double,double,double);\
  double (*pointToObject)(object_p,double,double);\
  int (*extent)(object_p,double *,double *,\
                double *,double *);

/*
 *  This is the shape class structure definition.
 *  All classes which are subclasses of shapeClass
 *  will consist of CLASS_PART and SHAPEC_PART as
 *  the first two parts of the class.
 */

typedef struct _shapeC
{
  CLASS_PART
  SHAPEC_PART
} shapeClass_t,*shapeClass_p;

/*
 *  given an instance of a class_p, these macros will
 *  access the shape class & the shape superclass
 */

#define getShapeClass(class) \
((class) ? (shapeClass_p)(class) : NULL)
#define getSuperClass(class) \
((class) ? ((shapeClass_p)(class))->superClass : NULL)

/*
 *  This is the shape object definition.  The class
 *  member is a 'handle' to the object's class and
 *  methods.
 *  The first member in any object is a pointer to
 *  the class structure which contains the methods
 *  for the object.
 *  The SHAPE_PART contains members specific to all
 *  geometric shape objects.
 *  minx,miny,maxx,maxy are coordinates of the
 *  bounding box for any object.
 */

#define OBJECT_PART \
  class_p      class;\
  void         *next;

#define SHAPE_PART \
  double       minx,miny;\
  double       maxx,maxy;

/*
```

```c
 *  shapeObject structure definition
 *  All shapeObjects and objects whose class is
 *  a subclass of shapeClass will consist of an
 *  OBJECT_PART and a SHAPE_PART as the first two
 *  parts of the object.
 */

typedef struct _shapeObject
{
  OBJECT_PART
  SHAPE_PART
} shapeObject_t,*shapeObject_p;

/*
 *  define some convenience macros for accessing the parts
 *  sanely.
 */

#define getObjectClass(object) \
((object) ? ((shapeObject_p)(object))->class : NULL )

/*
 *  symbols defined by shapeClass for method inheritance.
 *  These are dummy procedures.  The procedure names are
 *  used as function pointers in the appropriate subclass
 *  method slot instead of a 'real' procedure.  During
 *  class setup, shapeClass checks for these symbols, and
 *  if they are found, they are replaced by the actual
 *  shapeClass function pointer.
 */

extern shapeClass_t _shapeClass;

extern int inheritDisplay( object_p,window_p );
extern int inheritErase( object_p,window_p );
extern int inheritHighlight( object_p,window_p );
extern int inheritUnhighlight( object_p,window_p );
extern int inheritInsertdrag( object_p,window_p );
extern int inheritMovedrag( object_p,window_p,
                            double*,double* );
extern int inheritSizedrag( object_p,window_p,
                            double,double,
                            double*,double* );
extern int inheritRotatedrag( object_p,window_p,
                            double,double,double* );
extern int inheritMove( object_p,double,double );
extern int inheritResize( object_p,double,double,
                            double,double );
extern int inheritRotate( object_p,double,double,double );
extern double inheritPointToObject(object_p,
                            double,double);
extern int inheritExtent( object_p,double*,double*,
                            double*,double* );

#endif   /* SHAPECLASSP_DEFS */
```

**Listing Two**      lineClassP.h

```c
/*============================================================*/
/*
 *  Listing 2.  Private include file for lineClass.
 *
 *  lineClassP.h - lineClass extends shapeClass to include
 *                 methods for finding, dragging, and
 *                 moving the individual points in a
 *                 line.
 *
 *                 The structure for line objects extends
 *                 the shape object structure to include
 *                 a pointer to the list of points in the
 *                 line object.
 *
 *  (c) Copyright 1991, Forest W. Arnold
 *  All rights reserved.
 */
```

*(Listings 1-7 can be found on the AC's TECH Disk)*

```
/*=========================================================*/

#ifndef LINECLASSP_DEFS
#define LINECLASSP_DEFS

#include "globalDefs.h"
#include "shapeClassP.h"
#include "lineClass.h"

/*
 *  Line Class Part - line class adds methods for finding,
 *  dragging, and moving points
 */

#define LINEC_PART \
point_p (*findPoint)(object_p,double,double,double);\
int     (*dragPoint)(object_p,point_p,window_p,\
                     double *,double *);\
int     (*movePoint)(object_p,point_p,double,double);

/*
 *  The line class structure 'overloads' the shape class
 *  structure by adding LINEC_PART to the shape class
 *  structure.
 */

typedef struct _lineC
{
    CLASS_PART
    SHAPEC_PART
    LINEC_PART
} lineClass_t,*lineClass_p;

/*
 *  Line Object Part - line objects add a linked list of
 *  points to shape object.  The points contain the
 *  coordinates of the line.
 */

#define LINE_PART \
point_p points;
```

```
/*
 *  line objects are formed by 'overloading' the shape
 *  object structure.  LINE_PART is added to the end of
 *  the shape object structure.
 */

typedef struct _lineObject
{
    OBJECT_PART
    SHAPE_PART
    LINE_PART
} lineObject_t,*lineObject_p;

/*
 *  macro for accessing line class from a class_p
 */

#define getLineClass(class) \
((class) ? (lineClass_p)(class) : NULL)

/*
 *  external declarations for subclasses of lineClass.
 *  export actual class definition and inheritance
 *  symbol dummy procedures.
 */

extern lineClass_t _lineClass;

extern point_p inheritFindPoint(object_p object,
                                double x,double y,
                                double eps);
static int inheritDragPoint(object_p object,point_p point,
                                window_p window,
                                double *dx,double *dy);
static int inheritMovePoint(object_p object,point_p point,
                                double dx,double dy);

#endif   /* LINECLASSP_DEFS */
```

# Implementing an ARexx Interface in your C Program

By now, I am sure that many Amiga users have heard about ARexx and are well-aware of the capabilities of this macro language. I will not repeat the same information here. You will need some degree of familiarity with ARexx and a fairly good working knowledge of C to put the information presented in this article to good use. If you are not that familiar with ARexx or C, a study of those languages will help you understand this article better.

## Misconceptions

Before I began working with ARexx, I had some misconceptions concerning implementing an ARexx interface in my own programs. I believe other programmers also have some of these misconceptions. The following paragraphs will cover some of these possible misconceptions.

At first, I thought that it would be too difficult to add an ARexx interface to a program I had written. That turned out to be false, but that was mostly because of my programming style. There are programming styles that are conducive to adding an ARexx interface and there are programming styles that make it quite laborious. I am not suggesting that one style is correct and another style is incorrect—it is simply a matter of preference. However, when it comes to adding an ARexx interface, your programming style can save you programming time or increase it.

If you are a programmer that believes in the structured approach to programming, you should have little difficulty adding a ARexx interface to your programs. Structured programming uses separate functions to execute the subtasks that the program requires to perform its job. Each function is kept as small as possible to do its particular subtask. The structured programmer then includes calls to these functions within the C programming constructs such as the switch, if-else, do, while, and for statements. When adding an ARexx interface, it is just a matter of calling the proper function to perform the subtask required to fulfill the command requested by an ARexx message. This is the most efficient programming style when it comes to adding an ARexx interface. Some very good articles describing the structured programming approach to programming have been written by Paul Castonguay in previous *AC TECH* issues.

The alternative to the structured programming approach is to include the code that executes each subtask within the C programming constructs mentioned earlier. Each case within a switch construct contains the code required to perform a particular subtask rather than calling a function, for example. If this type of programming is your style, you will need to add functions to your program to execute when you start receiving commands in the form of ARexx messages. Redundant programming results since most of the instructions contained in the switch statement are going to be almost exactly the same as in the functions you have to add to execute the ARexx command requests. This is the least efficient programming style when it comes to adding an ARexx interface.

So you see, the difficulty you experience in implementing an ARexx interface in your program has much to do with your style, or approach to programming. If you are considering writing a program in which you would like to include an ARexx interface, then you might want to consider using a structured approach to programming. You should write each command so that you call it in exactly the same manner whether the source of the request is from your ARexx interface or from any other source.

Another misconception some public domain and shareware programmers might have is that they don't need to put an ARexx interface in their programs because not too many Amiga owners have ARexx. Some time ago that might have been true, but consider this: when Amiga owners upgrade to AmigaDOS 2.0, and almost every Amiga owner will inevitably want to, they will all have ARexx. Even now I suspect a large number of the Amiga owners have ARexx. If it were not so, the commercial software houses would not feel compelled to include ARexx interfaces in their programs. Public domain and shareware programs, some of which are every bit as good as their commercial counterparts, can also benefit from an ARexx interface. Imagine Amiga power users combining the use of your program with the commercial programs they have.

If you're concerned about the increase in the size of your program, you have no need to worry. Again, your style of programming determines this. If you write redundant code to implement each command your program can perform, your code size is going to increase much more than if you write functions that you can call from any location in your program. So, not only can your programming style determine the difficulty you are going to have including an ARexx interface, but it will also determine the size of the additional code you will need to write to have an ARexx interface in your program.

by David Blackwell

## Reasons for an ARexx Interface

I have already touched on some of the reasons why you might want to add an ARexx interface. If you are a public domain or shareware programmer, you obviously want your program to be used and enjoyed. Anything you can add to your program that will increase its usefulness will ensure that it does get used. An ARexx interface does increase the utility of your program. Even if you're simply writing a program for your own personal need, an ARexx interface will enhance its usefulness. However, if you write a program for your own personal need that you get considerable use out of, I would suggest introducing it into the public domain or releasing it as shareware.

Consider how your program could perform when combined with a high quality commercial, public domain, or shareware program. You could write an ARexx macro program that would unify the two programs. You could easily switch between the programs in memory making it appear as one unified programming environment. Staggering possibilities flood my imagination.

This type of program integration may also save you programming time. You may want to write a program that modifies the output of another program to suit your specific needs. If both programs have an ARexx interface, you are all set. You can then concentrate your work on the code to perform the modification and let the other program handle the initial work.

Sure, you could also accomplish this without each program containing an ARexx interface. This would require you to run one program and perform the work you want to do. After you had finished with the first program, you would run your own program to modify the output of the first program. That's not too much trouble, I guess, if you're used to that kind of juggling with your programs. However, I think that the idea of multi-tasking two or more programs under the control of a macro program is much more attractive. Both programs could be working on the same data to produce the final output that you want. This is what most Amiga owners are used to. This is the reason for a macro language like ARexx.

Another good reason for an ARexx interface is the control it gives the user over your program. Many software companies are going to great lengths in their advertising to inform Amiga owners and potential software purchasers when their products contain an ARexx interface. I believe they realize that the potential to tailor their program to the user's specific needs is very attractive to the user. Whenever I look at a new software package to buy, one of the first things I look for is an ARexx interface. I might not be able to make use of it right away, but I may eventually put it to good use. If you are producing a program with commercial or shareware potential, an ARexx interface may translate into monetary gains. If you are producing a program that you intend to release into the public domain, an ARexx interface may be just what you need to persuade people to try your program. Whether you are going for income or for the personal satisfaction of knowing that your program is being put to good use, an ARexx interface can help.

One final reason for an ARexx interface: support for the ARexx language is continuing to grow. Recently I have seen two new products that expand the ARexx language as released by Bill Hawes. One is an object-oriented programming tool and the other is an ARexx compiler. With the growth of support will most likely come an increase in interest by common Amiga owners. Soon, for a software package to be successful, I believe that an ARexx interface will be essential.

## Programming the Interface

The following description of the steps required to add an interface will give you what you need to get started. The interface can be as simple or as complex as you want to make it. The more complex of an interface you want, however, the more programming you will do to implement it. I will give you the basics and then it is up to you to determine how far you want to go with them.

The very first thing you must do is open the rexxsys.library through a call to the Exec's OpenLibrary function. You must put the return value from this function into a global variable named RexxSysBase. After you have opened the library, you have access to all the ARexx system functions. I will not cover these functions since that would take too long, and they are all well covered in Appendix C of the *ARexx User's Reference Manual*.

Getting a message port up and running should be your next priority. The easiest way to accomplish that is by using the Exec's CreatePort function (a complete description of this function can be found in the version *1.3 ROM Kernal Reference Manual: Libraries and Devices*, page 281 or on page B-5 of the version 1.2 Exec RKM). The CreatePort function will allocate all the necessary memory, add your port to the Exec's public port list, and allocates a signal bit for use with this message port. This message port can be considered the hub of your interface. Command requests, information requests, and replies to messages you sent out will come through this message port. All of your interaction with ARexx and any other ARexx host program or macro will go through this message port. Because this message port is vital to the execution of your interface, it is important that you keep up with its activity.

Most likely, your program will be receiving input from some source, and your program will probably be waiting for some event to occur before it can proceed with processing. This will most likely be in one form of the Wait function or another. Adding your port signal to the list of signals you are waiting on is simple and something you are probably already familiar with. If you are unfamiliar with the Wait function or how to set up to wait for multiple signals, you can refer to the main function of the Filer.c source code (Listing Two). Toward the end of the main function, I set up the signals I am going to wait on, and then in the while loop that follows, I wait on certain signals to occur. After my program wakes up, I test the return value from the Wait function to determine which signal woke up my program.

The alternative to waiting on signals to wake up your program is to create a busy loop and check your message port periodically to see if there is a message waiting. However, this type of programming is not considered appropriate in a multitasking operating system. This generally slows down the system and irritates many users. Therefore, I recommend the first procedure.

When your program wakes up and you have determined that the signal you allocated for your ARexx message port is the culprit, you need to get the new message and process it. Call the Exec's GetMsg function to get the address of the RexxMsg structure. (Refer to Figure One for the RexxMsg structure definition.) Now that you have the RexxMsg structure address you can determine what is being requested of your program.

---

*"If you are producing a program with commercial or shareware potential, an ARexx interface may translate into monetary gains."*

The request will be contained in the rm_Args[0] field, also known as ARG0, of the RexxMsg structure. This is the RexxArg structure that contains the command name (your program name) and the arguments that go with it. (Refer to Figure Two for the RexxArg structure definition.) The value contained in ARG0 can be treated like a pointer to a C style null-terminated string since that is basically what it is. All the fields in the RexxArg structure can be accessed at negative offsets from this pointer. However, this string is what contains all the information you are looking for. The first argument is the command name. You use this to determine which function to call. The remainder of the string should be parsed and sent to the function that executes the requested action. It is the information following the command name that you will need to extract from this string. All arguments are sent in string form. If you are expecting numeric arguments, you will need to use some of the C standard library functions to convert them to the format you require. There is a special condition that you need to be aware of. This occurs when the macro programmer requests that ARexx do the parsing for you.

Since you are just writing the host application and not every macro that could possibly call your program, you need to be aware that not every macro needs to call your program in exactly the same manner. To restrict the manner in which your program may be addressed by other programs is considered quite rude in some programming circles. What I am getting at is that a macro programmer can request that ARexx tokenize the command string prior to sending the RexxMsg structure to your program. The programmer does this by setting the RXFB_TOKEN command modifier flag bit in the rm_Action field of his or her RexxMsg structure. (Figure Three contains a complete listing of the Command modifier flag bits.) When ARexx receives a message structure with this bit set, it breaks up the ARG0 string into separate strings and creates a RexxArg structure for each one and places them in the rm_Args[0-15], ARG0-ARG15, fields in the order they appear in the command string. To correctly handle this situation, you need only to check the RXFB_TOKEN bit, and if it is set, get your arguments from the individual rm_Args[] fields and convert them as needed. To determine how many arguments the command string had, check the lowest nibble of the rm_Action field. This will be set to the number of arguments. If the command string has not been tokenized, then you will need to parse it as previously explained.

Once you have parsed the command and processed the request, you reply to the message. The important thing here is to report on the result of your processing so the program that sent the message can know what action to take. If all went well, set the rm_Result1 field to RC_OK. (See Figure Four for the ARexx return codes.) The rm_Result2 field is used to return a pointer to the result string if the RXFB_RESULT command modifier bit is set. The return string must be in the form of a RexxArg string. A result string should be returned only if requested and no errors occurred during you processing. If you were unsuccessful in your attempt to process the request, you set the rm_Result1 field with severity level of the error and the rm_Result2 field should be set to zero.

## Figure One
### RexxMsg Structure Definition

```
struct RexxMsg {
    struct Message rm_Node;   /* Exec message structure      */
    APTR rm_TaskBlock;        /* pointer to global structure */
    APTR rm_LibBase;          /* library base                */
    LONG rm_Action;           /* command code                */
    LONG rm_Result1;          /* primary result             */
    LONG rm_Result2;          /* secondary result           */
    STRPTR rm_Args[16];       /* argument block (ARG0 - ARG15) */
    struct MsgPort *rm_PassPort; /* forwarding port           */
    STRPTR rm_CommAddr;       /* host address                */
    STRPTR rm_FileExt;        /* file extension              */
    LONG rm_Stdin;            /* input stream                */
    LONG rm_Stdout;           /* output stream               */
    LONG rm_avail;            /* future expansion            */
};
```

## Figure Two
### RexxArg Structure Definition

```
struct RexxArg {
    LONG ra_Size;       /* total allocated length */
    UWORD ra_Length;    /* length of string       */
    UBYTE ra_Flags;     /* attribute flags        */
    UBYTE ra_Hash;      /* hash code              */
    BYTE ra_Buff[8];    /* buffer area            */
};
```

## Table One
### Command Modifier Flag Bits

| | |
|---|---|
| RXFB_NOIO | Suppress I/O inheritance |
| RXFB_RESULT | Result string expected |
| RXFB_STRING | Program is a string file |
| RXFB_TOKEN | Tokenize the command line |
| RXFB_NONRET | A non-return message |

## Figure Three
### ARexx Return Codes for General Use

| | | |
|---|---|---|
| RC_FAIL | -1 | Something's wrong |
| RC_OK | 0 | Success |
| RC_WARN | 5 | Warning only |
| RC_ERROR | 10 | Something's wrong |
| RC_FATAL | 20 | Complete or severe failure |

## Figure Four
### Rexx Data Structure

```
struct RexxData {
        struct MsgPort RexxPort;
        CHAR *Exten;
        APTR Func;
        struct *RexxLib;
        APTR Error;
        APTR Result;
        ULONG RexxMask;
        ULONG UserData;
        /* associated command list for your program */
        struct CmdEntry AsscList[NUMCMDS];
        };
```

Not every message received at your message port is a request for some sort of action. With only one exception, if you send any messages to the ARexx resident process, you can expect to eventually receive reply messages. When these replies start rolling in, you need to be able to distinguish between them and requests for information or command requests. This is a fairly straightforward procedure. You check to see if the message you just retrieved is the type NT_REPLYMSG. This will be in the UBYTE ln_Type field in the rm_Node message structure in your RexxMsg message structure. The test could look like this:

```
if (MyMessage->rm_Node.mn_Node.ln_Type == NT_REPLYMSG)
```

When you do determine that it is a reply message, check for possible error conditions and handle any that appear. If all went well and you requested a return string, you need to extract the return string, delete any RexxArg's and then delete the RexxMsg itself. If you did receive a return string, you are responsible to delete this RexxArg structure when you are done with it.

Another way to distinguish between requests and replies is to set up a separate message port for replies only. Put the address of this message port in the mn_ReplyPort field of the RexxMsg message structure.

```
MyMessage->rm_Node.mn_ReplyPort = MyReplyPort;
```

Now, when the message returns to you it will come back to your reply port, and you can have a customized function to handle all reply messages received at this port. Your other ARexx message port will only receive requests of one kind or another and can have its own function to send off those request to your program. The way you handle your message traffic is up to you. Each way has its pros and cons.

Earlier I referred to the fact that some messages you send out may not come back in the form of a reply message. If you set the RXFB_NONRET command modifier flag bit in the RexxMsg rm_Action field, you will never hear from that message again. Maybe that is a good idea. What happens, however, if the action you requested is not successfully completed? You may never know. Live on the edge; try it.

So far I have just written about commands received from other programs instructing your program on what to do. However, you can direct other programs through your ARexx interface also. You can get ARexx macros running. You can run other host applications. You can exchange information with other host applications. You can even combine these possibilities in new and imaginative ways. All of your communications with exterior programs, however, should go through the ARexx resident process. That requires you to get an ARexx message put together with the proper initialization of the fields that are required to get the message to the proper destination.

The first thing you have to do, of course, is allocate some memory for the RexxMsg structure. I generally request that the memory be cleared as it is allocated (ex., MyRexxMsg = AllocMem(sizeof(struct RexxMsg), MEMF_CLEAR) ). This has the effect of initializing all the unused RexxMsg fields for you. All you have to do then is set up the fields you are using. The rm_Action field is extremely important. This is the action code (command) you are sending to ARexx. There are a number of these action codes, but we will focus on the command-level invocation code (RXCOMM). As previously covered, there are some command modifier flag bits that can be set in the rm_Action field to further tailor the action code. These command modifier flag bits can be combined to finely tweak

the command execution. However, setting the RXFB_NONRET and the RXFB_RESULT bits at the same time may have some serious side effects. The first bit instructs ARexx not to return the message to you and the second bit requests that a result string be returned in the message structure that you just requested not be returned. (What a paradox for ARexx–I am not exactly sure what would happen.) Next you place your message port's address in the MN_REPLYPORT and rm_PassPort fields of your message structure. Then you put the pointer to your message port's name in the MN_NAME and rm_CommAddr fields. Optionally, you can supply a file extension for ARexx to use when looking for the command name you sent to be executed by placing a pointer to a file extension string in the rm_FileExt field, and you can send your program's default input and output filehandles by placing them in the rm_Stdin and rm_Stdout fields respectively.

As described, the task of implementing an ARexx interface seems quite imposing. There are quit a few minor details that must be attended to. The task is simple enough, but allocating and releasing memory for ARexx messages and argstrings can take its toll. Creating a message port for your interface to use, monitoring the message port's activity, parsing the command strings, distinguishing between command requests and reply messages, all the tasks associated with sending a command to ARexx and finally deleting your message port when you are done with it can be a nuisance. Don't get me wrong, an ARexx interface is worth all you go through to have it. On the other hand, if most of the tedious work was done for you, wouldn't you want to take advantage of that?

## Rexxapp.library Answers the Call

That is exactly what Jeff Glatt of Dissidents Software has done. He has put together a library of functions that will take care of all the mundane tasks associated with an ARexx interface. This is the rexxapp.library.

This library automatically handles all of the message traffic associated with your ARexx message port. It accepts asynchronous ARexx messages and sends both synchronous and asynchronous messages to ARexx on behalf of your program. This means that you can control other programs or be controlled by other programs through this library.

The rexxapp.library allocates and frees the memory for all the ARexx structures it uses. You no longer work directly with the ARexx resident process. You don't have to supply the routines to handle a message port. You give the library a message port name, and it opens the message port. The library creates all the ARexx messages and sends them out at your request. It also distinguishes between commands received and replies to messages you sent out. It can even call separate routines based on whether a reply message returns a result string or an error condition. All this processing power is packed into just seven functions.

This limited number of functions means that with very little programming you can have an ARexx interface. To take advantage of the library, first set up the RexxData structure defined by Jeff Glatt for use by the library. (See Figure Five for the RexxData structure.)

## RexxData Structure

The RexxData structure is a required argument for every function in the library. It is a variable length structure. The size of the structure varies with the number of commands your program recognizes. You supply the number of commands in the NUMCMDS variable in your header file. This variable must be defined before you declare the RexxData structure type. You must initialize the Exten, Func, Error, Result and AsscList[] fields of your RexxData structure before you can use it.

The Exten field holds a pointer to the null-terminated string to use as the filename extension to affix to commands received by the library that are not in your associated command list. This situation can happen if the ARexx address command, followed by your port name, precedes a command the ARexx interpreter does not understand. The ARexx interpreter will package this command up as an argstring, place a pointer to this argstring in the ARG0 field of a RexxMsg structure and send this message to your program for processing. If this command is actually the name of an ARexx macro, and not an internal command of your program, the library adds the filename extension to it and sends the message back to the ARexx resident process as a command invocation request. ARexx will then run the macro for your. With this feature, you can use ARexx macros in the same fashion as internal commands. If you don't really want an extension but you would like to take advantage of this feature, put a pointer to a nulled string in this field. The library will pass the command on to ARexx exactly as received. By placing a zero in this field you turn this feature off. That means that when the library encounters a message sent to your message port, that does not contain a command in your associated command list, it will be returned to ARexx with an error value of 30 in the rm_Result1 field.

The Func, Error and Result fields hold pointers to functions. You write these functions, and the library calls them to handle certain situations. Writing these three functions is probably the most work you will do in setting up your interface using the rexxapp.library.

The Func field holds a pointer to the function you want the library to call when a message arrives at your message port that has a command in the argstring that is also found in your associated command list. Dispatch is the name of my function. The library calls this function only when your program calls the ReceiveRexx library function (more on the ReceiveRexx function later). The dispatch function receives four arguments: a pointer to the message that prompted the call, the pointer to the function to call to execute the requested command, a pointer to the argstring stripped of the command name, and a pointer to your RexxData structure. This function should set up any arguments or variables that the requested function needs and then calls the function. When the function you called returns, you need to set up the result fields using the message pointer you received in your dispatch function's first argument. The result fields are set up differently depending on whether the command was executed successfully or not. The library's SetupResults function gives you a convenient way to set up the result fields (again, more on the SetupResults function later). After your dispatch function is done, you return either a one or a zero. A return value of one tells the library to reply to the message. A return value of zero tells the library that your program is going to reply to the message. Unless you absolutely need to hold onto the message for some reason, you should let the library reply to the message. One side effect of holding onto the message is that the ARexx script that sent the message is suspended, put to sleep in other words, until you reply to the message.

The Error field holds a pointer to the function for the library to call whenever any message that your program sends out returns with an error. Most likely this function will just print the error message to the screen, in one manner or another, to let the user know that an error occurred. If you don't want to do anything with error returns, you must still provide a dummy routine. Your error function receives four arguments: the error code, a pointer to the error string, a pointer to the message that returned with an error and a pointer to your RexxData structure. Do not alter the error string.

The Result field holds a pointer to the function for the library to call whenever any message your program sends out returns after a successful completion. What your functions does with the result string depends on which function in your program sent the

message in the first place. If you never request a result string to be returned, you still need to provide a dummy routine. Your result function receives four arguments: the result code, a pointer to the result string, a pointer to the message that has returned and a pointer to your RexxData structure. Do not alter the result string.

You can find the prototypes to my functions in Listing One, my header file, and you can see the actual code in Listing Two, my source code. The names of my functions are: dispatch, error_rtn and process_rtn. In this article, my dispatch routine is a scaled down routine. In Part Two of this article, I will add more to this routine to make it a little more useful. My error routine, error_rtn, simply displays an autorequester to the window to inform the user what has happened. My return function, process_rtn, is just a dummy routine in this program. I never request a result string. However, in the completed program that will accompany Part Two of this article, this routine may change if needed.

The last field you initialize in the RexxData structure is the AsscList[] field. This is an array of CmdEntry structures. Each command entry structure contains a name string of a command your program recognizes and a pointer to the function that is associated with this command string. The variable NUMCMDS defines the number of commands in your list. The last entry in the list must be a NULL entry so the NUMCMDS variable must actually equal the number of commands your program recognizes, plus one. All the commands must be in lower-case letters with no imbedded spaces. The library does not require the complete command string in order to

match a command request to the command list entry. If one of your commands in your list is open, then the string ope will also result in a match. Be sure to make all of your command names unique. Any command of yours that matches an ARexx instruction will be sent to the ARexx interpreter instead of to your program.

Now let's go over the functions available for you to use in the library. There are only seven functions to cover: SetRexxPort, ReceiveRexx, SetupResults, FreeRexxPort, SendRexxCmd, SyncRexxCmd and ASyncRexxCmd.

## SetRexxPort

The SetRexxPort function must be called first. This function does all your interface set up. You pass it a pointer to your port name and a pointer to your RexxData structure. This function returns a signal bit mask assigned to your message port. You can use this mask in the Exec Wait function to wait for activity at this message port. The port name string buffer should be one byte larger than the actual name. This function appends a number between 2-9 on the end of your port name to resolve any conflicts with port names already used. The library can support only up to nine copies of your program running at once. This function will initialize the remaining fields in your RexxData structure. You call this function only once. This function returns a zero if it was unsuccessful. Once you have successful set up, you are ready to continue.

## ReceiveRexx

The ReceiveRexx function is the workhorse of the library. When a message arrives at your message port, your program is awakened by a return from the Wait function. When you test the return value and determine that the signal from your ARexx message port caused your program to become active, you call the ReceiveRexx function. The ReceiveRexx function takes care of all the messages waiting at your message port. It can tell the difference between commands received from other programs and replies to messages you initiated. If the message is a command, ReceiveRexx will call your dispatch function. If the message is a reply, ReceiveRexx checks the rm_Result1 field of the message structure to see if the request was executed successfully. ReceiveRexx calls your result function if everything went well; otherwise, it calls your error function. This function frees the memory used by messages you sent out and replies to all messages that your message port receives. The only time it will not reply to a message is if your dispatch function returns a zero. Before your dispatch function returns, however, it should have called the SetupResls function.

## SetupResults

The SetupResults function is used by you to communicate to the initiating program whether its request was successfully completed. You provide a primary result value, a secondary result value, a pointer to a return string, a pointer to the message structure that caused the action and a pointer to your RexxData structure. If the request was successful, pass a zero as both the primary and secondary result values, and you pass a pointer to a result string if the RXFB_RESULT command modifier bit is set. If the RXFB_RESULT bit is not set or you don't have a string to return, you supply a NULL as a result string pointer. You return the pointer to the message structure that ReceiveRexx supplied. You return error values fitting the error severity level in both the primary and secondary result values and a NULL as the result string pointer if you did not successfully complete the request. This function does not return a value.

## FreeRexxPort

The FreeRexxPort function provides a clean exit for your ARexx interface. Once you have executed the SetRexxPort function it is absolutely necessary to execute this function when your program is done. This functions takes a lot of work off your hands: it replies to any messages remaining in your queue, it closes your ARexx message port, it frees any resources it used, and it closes the ARexx system library. If any of the messages you sent have not returned yet, this function will not return until every one of them is accounted for.

## Communication Routines

The next three library functions handle communications with ARexx. Two of these functions are high level functions; one synchronous and one asynchronous. You pass fewer arguments to the higher level functions; therefore, they are easier to use. You will most likely get the most use out of the higher level functions. The one low-level function is used when you need the maximum control of the communication process that is possible with the library. You pass twice as many arguments to the lower-level function as compared to the higher level functions. The main attraction of the lower-level communication function is that you have a chance to manipulate the RexxMsg structure before it is sent, and you get the first look at it when it returns. The document file that is included with the library alludes to the fact that this function is both synchronous and asynchronous. During my testing, I was unable to get it to function synchronously. I am continuing to try to work this problem, because there are definitely times when you want to wait for your request to finish executing before you continue with your processing.

## SyncRexxCmd and ASyncRexxCmd

SyncRexxCmd and ASyncRexxCmd are the higher level functions for synchronous and asynchronous communications respectively. Other than the basic difference between synchronous and asynchronous communications, these functions are very similar. The SyncRexxCmd function requires three arguments: a pointer to the command string, a pointer to the message structure that caused the action, and a pointer to your RexxData structure. The ASyncRexxCmd function requires only the command string pointer and the pointer to your RexxData structure as arguments. Both functions return a pointer to the message structure it created to communicate with ARexx. However, if either function cannot successfully fulfill your request, it returns a zero value, and it places a pointer to a null-terminated error message in the global variable RexxErrMsg. Another difference in operation is that the SyncRexxCmd function replies to the message that you passed the pointer for. These functions are mainly provided for ease of use rather than the maximum amount of control of the communication process.

## SendRexxCmd

The SendRexxCmd function, as the lowest-level communication function of the library, gives you the most control over the communication process. Unlike the previous two functions where the most control you had was to send a command string, with the SendRexxCmd function, you provide the value for the rm_Action field of the message structure, a pointer to a message initialization function, a pointer to a command string, a pointer to a returned message function, a pointer to the message that caused this action, and a pointer to your RexxData structure. Listing One contains the prototype for the SendRexxCmd function. The most promising possibilities come from the two function pointers you provide.

You can use the first function to modify or initialize any field in the message structure before it is sent out. The SendRexxCmd function performs a default initialization before passing the message structure to your routine. Of particular interest to us are the ARG1-ARG13, rm_PassPort, rm_CommAddr, Stdin and Stdout fields.

The SendRexxCmd function initializes the ARG0 field with the pointer to the command string you passed as your third argument in the function call, and it uses ARG14 and ARG15 for its own purposes. This leaves you the ARG1-ARG13 fields to use however you see fit. You can use these fields to pass additional argstrings, C style terminated strings, numeric values, or any other value that you can fit into a STRPTR (long) data type. However, you are responsible to free any memory allocated to pass argstrings or C style null terminated strings. You can free this memory in your error and result routines described earlier or in the returned message function for which you pass a pointer to the SendRexxCmd function. It is important to remember that if you use some of the ARG1-ARG13 fields to pass values other than argstrings, the program that receives the message must be prepared to process those values. This eliminates ARexx macros because they expect argstrings only.

The rm_PassPort field holds a message port address. ARexx will parse the ARG0 value to extract the command name. ARexx will then search for a program by that name. If the program is not found, ARexx will pass the message to the message port address in the rm_PassPort field. The command name in ARG0 may not be a program name at all but a command that the other program, whose message port address you put in rm_PassPort, recognizes. This provides a convenient way for you to pass commands to another program using the ARexx resident process as an intermediary. If this field is a NULL and ARexx can not find the program, then ARexx returns the message with a "Program not found" error message.

The rm_CommAddr field on the other hand provides a way to override the default initial host address. This is a null-terminated string and is "REXX" by default. The ARexx User's Reference Manual states, "The host address is the name of the message port to which commands will be directed ...." You can redirect that command message traffic by supplying your message port name as the default address. The macros you run with your message port name as the default host address no longer need to execute the ARexx address instruction to direct commands to your program. They come to your program automatically now. This is convenient if your program supports multiple instances of itself. Each instance of your program will have a unique message port name. This message port name will be used as the default host address by each instance. That allows each instance of your program to run exactly the same macros, and the commands issued by those macros will be routed automatically to the correct instance of your program by use of the default host address value. Still a little vague? I will try to clarify it further with an example.

```
Example Macro
/* required ARexx comment */
say 'Example macro running.'
  'save' /* command for your program */
exit
```

That macro is easy enough to understand. First it prints a message alerting the user that it is running, sends a save command and then quits. Without the ARexx address command, the save command is sent directly to the ARexx resident process because the default host address is "REXX". If this macro is executed with your host address name supplied as the default, the save command would be sent directly to your program. With multiple instances of your program running, each with unique host address names, the save command will be sent to the proper instance of your program by use of the default host address name when they execute this macro. Using this feature allows you to write macros that can be used by any program that supplies the proper default host address, and multiple programs can execute these macros simultaneously.

The last two fields of interest, rm_Stdin and rm_Stdout, allow you to redirect the input and output streams. If your program, for one reason or another, has no default input and output streams, it will be necessary for you to supply values for these fields if your macro prints any information to the screen or prompts the user for any information.

The second function you supply to the SendRexxCmd function gets the first look at the message structure when it returns. You can use this routine to extract any special values you expect to be returned. You may also need to free some memory used by argstrings you sent. This function can also be called after a fatal error, so it is important to test return values before using them to ensure that they are safe to use. This function is passed a pointer to the RexxMsg you sent out, the rm_Result1 value and the rm_Result2 value. The document file that comes with the library says that if no routine is supplied then the SendRexxCmd function executes asynchronously. Therefore, I assume that if a routine is supplied, the SendRexxCmd function executes synchronously. However, even when I supplied a routine, the SendRexxCmd function continued to execute asynchronously. I am continuing to test this function.

## Conclusion

Well, that is a fairly detailed description of the the rexxapp.library written by Jeff Glatt of Dissidents Software. The program included with this article is a completely functioning program with just a shell of an ARexx interface. In Part Two of this article we will complete the interface. As the program is, it is completely useless except as a good test bed to explore the operation of the rexxapp.library. If you are ingenious, you can modify it a little to test every function in the library. I have done that myself. The completed program will be a program that you can get some use out of.

If you examine the source code in Listing Two, you will get a better idea of some of the material I presented in my description of the rexxapp.library. Because it is just an interface shell rather than a fully functioning interface, the RexxData structure includes only the commands available in the program's project menu.

When we complete the interface, we will give it access to more commands, some of which are available only through the ARexx interface. For some of the commands, we can write ARexx macros to execute them to reduce the size of the memory-resident host application. I encourage you to take a close look at the program to see what potential it has for you. This program is intended to be running in the background at all times for quick access to information stored in small database type files. You access it through the Alt-Ctrl-f key combination, pull up the information that your are looking for, and then put the program back to sleep until you need it again. All it needs to reach this goal is more file access commands and some display commands so you can see what is in the files. This is what we will be adding in the next part of the article. If you think you could get some use from a program like this, you can send me suggestions and ideas for commands you would like to see included in it. You can leave me a message on GEnie using the e-mail address, D.Blackwell1.

```
/* Filer.h
 * Header file for the Filer program.  This file should be pre-compiled to
 * save compile time.
 */

/* .... INCLUDES .... */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <functions.h>
#include <exec/memory.h>
#include <exec/execbase.h>
#include <libraries/dosextens.h>
#include <devices/input.h>
#include <devices/inputevent.h>
#include <rexx/rexxio.h>
#include <rexx/rxslib.h>
#include <rexx/errors.h>
#include <intuition/intuitionbase.h>

/* .... DEFINES .... */
#define DOS_REV 33
#define GRAPHICS_REV 33
#define INTUITION_REV 33
#define REXX_REV 0L
#define REXXAPP_REV 0L
#define BCOL 0
#define FCOL 1
#define RAW 1101
#define KEY 1102
#define SIZE 1103
#define NAME 1104
#define NUMCMDS 5
#define MAXFIELDS 12
#define REXXAPPNAME "rexxapp.library"
#define XORIGIN 0
#define YORIGIN 10
#define XLENGTH 80
#define YLENGTH 10
#define XDELTA 235
#define YDELTA 45
#define FWINDOWIDCMP CLOSEWINDOW|MENUPICK|ACTIVEWINDOW|INACTIVEWINDOW
#define FWINDOWFLAGS WINDOWCLOSE|ACTIVATE|NOCAREREFRESH
#define NFWINDOWIDCMP CLOSEWINDOW|GADGETUP
#define NFWINDOWFLAGS WINDOWCLOSE|ACTIVATE|NOCAREREFRESH|WINDOWDEPTH|WINDOWDRAG

/* .... STRUCTURE DECLARATIONS .... */
struct CmdEntry {
                    char *CmdString;
                    APTR CmdHandler;
            };

struct RexxData {
                    struct MsgPort RexxPort;
                    char *Exten;
                    APTR Func;
                    struct RxsLib *RexxLib;
                    APTR Error;
                    APTR Result;
                    ULONG RexxMask;
                    ULONG UserData;
                    struct CmdEntry AsscList[NUMCMDS];
            };

struct definition {
                    struct Node defNode;
                    UBYTE type;
                    UBYTE size;
                    char Name[14];
            };

/* .... FUNCTION PROTOTYPES .... */
char *initrequest(struct RexxMsg *, struct RexxData *);
void Close_Libs(void);
void freedeflist(void);
void inform(struct Window *);
void DeleteArgstring(STRPTR);
void close_input_handler(void);
void DeleteRexxMsg(struct RexxMsg *);
void FreeRexxPort(struct RexxData *);
void ReceiveRexx(struct RexxData *);
void error_rtn(ULONG, char *, struct RexxMsg *, struct RexxData *);
void process_rtn(ULONG, char *, struct RexxMsg *, struct RexxData *);
void SetupResults(LONG, LONG, STRPTR, struct RexxMsg *, struct RexxData *);
BOOL closeit(void);
BOOL wakeup(void);
BOOL newfile(void);
BOOL activate(void);
BOOL openfile(void);
BOOL savefile(void);
BOOL quitprog(void);
BOOL fileinfo(void);
BOOL dispinfo(void);
BOOL printfile(void);
```

```
BOOL aboutprog(void);
BOOL Open_Libs(void);
BOOL savefileas(void);
BOOL processmenu(USHORT);
BOOL set_up_deflist(void);
BOOL getafilename(UBYTE *);
BOOL intuirequest(UBYTE *);
BOOL getreturn(struct MsgPort *);
BOOL requestname(struct MsgPort *);
BOOL install_input_handler(struct MsgPort *);
BOOL extractrtn(struct RexxMsg *, LONG, LONG);
BOOL addlistitem(char *, unsigned int, UBYTE);
BOOL dispatch(struct RexxMsg *, BOOL (*fpt)(void), char *, struct RexxData *);
ULONG SetRexxPort(STRPTR, struct RexxData *);
STRPTR CreateArgstring(STRPTR, ULONG);
struct RexxMsg *ASyncRexxCmd(STRPTR, struct RexxData *);
struct RexxMsg *CreateRexxMsg(struct RexxMsgPort *, STRPTR, STRPTR);
struct RexxMsg *SyncRexxCmd(STRPTR, struct RexxMsg *, struct RexxData *);
struct RexxMsg *SendRexxCmd(LONG, APTR, STRPTR, APTR, struct RexxMsg *,\
            struct RexxData *);
struct Window *Open_Window(SHORT, SHORT, SHORT, SHORT, ULONG, ULONG, UBYTE *,\
            struct Screen *);
extern void myEventHandler(void);

/* .... AREXX PRAGMAS .... */
#pragma amicall(RexxSysBase, 0x90, CreateRexxMsg(a0,a1,d0))
#pragma amicall(RexxSysBase, 0x7e, CreateArgstring(a0,d0))
#pragma amicall(RexxSysBase, 0x84, DeleteArgstring(a0))
#pragma amicall(RexxSysBase, 0x96, DeleteRexxMsg(a0))
```

```
/* Filer.c */
/* Manx 5.0d source */
/* cc -mc -md -sa -sb -sn -sp -sr -ss Filer.c */
/* ln Filer.o RexxInterface.o c.lib */

#include "Filer.h"

/* .... GLOBALS .... */
struct      GfxBase *GfxBase = NULL;
struct      DosLibrary *DosBase = NULL;
struct      IntuitionBase *IntuitionBase = NULL;
struct      RexxBase *RexxBase = NULL; /* rexxapp.library pointer */
struct      RxsLib *RexxSysBase = NULL; /* ARexx system library */
UBYTE       undobuf[11]; /* Undo buffer for all string gadgets */
int         fieldcount = 0; /* keeps count on number of definitions in a record */
char        *success_str; /* used to return a string upon successful execution */
char        *error_str; /* used to return an error string for display to user */
char        FileName[110]; /* maximum file name length my filereq can return */
BPTR        fileptr = NULL; /* the active file handle */
BOOL        sleeping = FALSE; /* These next three variables describe the current */
BOOL        window_inactive = FALSE; /* state of the Filer Window throught the */
BOOL        all_done = FALSE; /* execution of the program */
ULONG       EventSignal; /* wake-up signal for event handler */
ULONG       PortSignal; /* IDCMP signal */
ULONG       RexxSignal; /* ARexx port signal */
ULONG       waitsigs; /* Signals that could wake-up the program */
struct      Window *myWindow;
struct      List *deflist = NULL;
struct      Interrupt *myHandler;
struct      IOStdReq *myRequest;
struct      MsgPort *myPort;
struct      Task *myTask; /* pointer to Filer task structure */

struct RexxData myRD = {
                                (NULL),
                                (char *) "fil",
                                (APTR) dispatch,
                                NULL,
                                (APTR) error_rtn,
                                (APTR) process_rtn,
                                NULL,
                                NULL,
                                { { "new",   (APTR)&newfile },
                                  { "open",  (APTR)&openfile },
                                  { "save",  (APTR)&savefile },
                                  { "quit",  (APTR)&quitprog },
                                  { (char *) 0,        (APTR) 0 } }
                        };

struct TextAttr MyFont = {(STRPTR)"topaz.font",TOPAZ_EIGHTY,\
            FS_NORMAL,FPF_ROMFONT};

/* Include all the Menu, Gadget, Requester and associated definitions */
#include "Filer.pw" /* Modified PowerWindows output file */

void main (void)
{
    LONG myClass;
    WORD myCode;
    ULONG rtnsigs;
    struct RexxMsg *myRexxMsg;
    struct IntuiMessage *myMessage;
    static char PortName[6] = "AFiler"; /* my ARexx port name */
```

```c
if (!Open_Libs())
    goto main_exit; /* catastrophic failure, can't continue */

if (!(myPort = CreatePort("Filer", NULL)))
{ /* a message port is critical to the program's operation */
    puts("Unable to create a message port.  Exiting!");
    goto main_exit;
}

if (!(myWindow = Open_Window((SHORT) XORIGIN, (SHORT) YORIGIN, \
        (SHORT) XLENGTH, (SHORT) YLENGTH, NULL, FWINDOWFLAGS, \
        (UBYTE *)"Filer", (struct Screen *)0)))
{
        puts("Insufficient memory for the Filer Window.  Exiting!");
        goto main_exit;
}

myWindow->UserPort = myPort; /* Add the message port then modify the */
ModifyIDCMP(myWindow, FWINDOWIDCMP); /* window's IDCMP flags */

if (RexxBase) /* if RexxBase is not 0 then the library is available */
{
        if (!(RexxSignal = SetRexxPort((STRPTR)PortName, &myRD)))
        {
            error_str = "Can't open Rexx message port, no ARexx interface.";
            inform(myWindow);
            CloseLibrary((struct Library *) RexxBase); /* Can't open a port */
            RexxBase = (struct RexxBase *)0; /* so close the library */
        }
}

myTask = FindTask(0); /* The next two globals are intended solely for */
EventSignal = (1 << AllocSignal(-1)); /* used by my event handler */
PortSignal = (1 << myPort->mp_SigBit);

/* This assignment works correctly regardless of RexxSignal's value */
waitsigs = (PortSignal|EventSignal|RexxSignal);

install_input_handler(myPort); /* start my input handler */

SetMenuStrip(myWindow, MenuList);

while (!all_done)
{
        rtnsigs = Wait(waitsigs);
        if (rtnsigs & EventSignal)
        { /* if the program is sleeping or the window is inactive and it */
            /* receives an input event signal, then wake up or activate the */
            /* window and process the user's request */
            if (window_inactive)
                    activate();
            if (sleeping)
                    wakeup();
        }
        if (!!sleeping) && rtnsigs & PortSignal)
        { /* Process all IDCMP messages that arrive at my window's message */
            /* port. */
            while (myMessage = (struct IntuiMessage *)GetMsg(myPort))
            {
                    myClass = myMessage->Class;
                    myCode = myMessage->Code;
                    ReplyMsg((struct Message *)myMessage);
                    switch(myClass)
                    {
                      case CLOSEWINDOW:
                            closeit();
                            break;
                      case MENUPICK:
                            all_done = processmenu(myCode);
                            break;
                      case ACTIVEWINDOW:
                            sleeping = FALSE;
                            window_inactive = FALSE;
                            break;
                      case INACTIVEWINDOW:
                            sleeping = FALSE;
                            window_inactive = TRUE;
                            break;
                      default:
                            error_str = "Unexpected class of Intuition Message!";
                            inform(myWindow);
                    }
            }

            if (RexxBase && (rtnsigs & RexxSignal))
            { /* If ARexx is available and we receive a signal from our ARexx */
                /* message port, call the rexxapp.library ReceiveRexx() function */
                /* to process all the outstanding ARexx messages. So simple!*/
                ReceiveRexx(&myRD);
            }
        }
}

close_input_handler(); /* shut down input handler */

main_exit:
if (myWindow)
{
    if (myWindow->MenuStrip)
        ClearMenuStrip(myWindow);
```

```c
        myWindow->UserPort = NULL;
        CloseWindow(myWindow);
}

if (myPort)
{ /* reply to all outstanding messages and delete the message port */
    while (myMessage = (struct IntuiMessage *)GetMsg(myPort))
            ReplyMsg((struct Message *)myMessage);
    DeletePort(myPort);
}

if (fileptr)
    Close(fileptr);

if (deflist)
    freedeflist();

if (RexxSignal)
    FreeRexxPort(&myRD);

Close_Libs();

return;
}

BOOL Open_Libs(void)
{
    BOOL success;

    success = TRUE;

    /* The first three libraries are critical to the program's operation */

    if (!(GfxBase = (struct GfxBase *)\
            OpenLibrary("graphics.library", GRAPHICS_REV)))
    {
        puts("graphics.library could not be loaded.");
        success = FALSE;
    }

    if (!(DosBase = (struct DosLibrary *)OpenLibrary("dos.library", DOS_REV)))
    {
        puts("dos.library could not be loaded.");
        success = FALSE;
    }

    if (!(IntuitionBase = (struct IntuitionBase *)\
            OpenLibrary("intuition.library", INTUITION_REV)))
    {
        puts("intuition.library could not be loaded.");
        success = FALSE;
    }

    /* If either of the next two libraries don't open correctly, the Open */
    /* Library function is still successfull. We just won't have an ARexx */
    /* interface to work with. */

    if (!(RexxBase = (struct RexxBase *) OpenLibrary(REXXAPPNAME, REXXAPP_REV)))
            puts("rexxapp.library not found, no ARexx port available.");

    if (!(RexxSysBase = (struct RxsLib *) OpenLibrary(RXSNAME, 0L)))
            puts("rexxsyslib.library not found, no ARexx port available.");

    return(success);
}

void Close_Libs(void)
{
    if (GfxBase)
            CloseLibrary((struct Library *) GfxBase);

    if (DosBase)
            CloseLibrary((struct Library *) DosBase);

    if (IntuitionBase)
            CloseLibrary((struct Library *) IntuitionBase);

    if (RexxBase)
            CloseLibrary((struct Library *) RexxBase);

    if (RexxSysBase)
            CloseLibrary((struct Library *) RexxSysBase);
}

struct Window *Open_Window(SHORT leftedge, SHORT topedge, SHORT width,\
        SHORT height, ULONG idcmp, ULONG flags, UBYTE *title,\
        struct Screen *scrn)
{
        struct NewWindow NewWindow;

        NewWindow.LeftEdge = leftedge;
        NewWindow.TopEdge = topedge;
        NewWindow.Width = width;
        NewWindow.Height = height;
        NewWindow.DetailPen = (UBYTE) BCOL;
        NewWindow.BlockPen = (UBYTE) FCOL;
        NewWindow.IDCMPFlags = idcmp;
        NewWindow.Flags = flags;
```

```c
        NewWindow.FirstGadget = (struct Gadget *) 0;
        NewWindow.CheckMark = (struct Image *) 0;
        NewWindow.Title = title;
        NewWindow.Screen = (struct Screen *) 0;
        NewWindow.BitMap = (struct BitMap *) 0;
        NewWindow.MinWidth = width;
        NewWindow.MinHeight = height;
        NewWindow.MaxWidth = (USHORT) 0;
        NewWindow.MaxHeight = (USHORT) 0;
        NewWindow.Type = WBENCHSCREEN;

        if (scrn) /* The user supplied a screen so make some changes */
        {
                NewWindow.Screen = scrn;
                NewWindow.Type = CUSTOMSCREEN;
        }

        return(OpenWindow(&NewWindow));
}

BOOL activate(void)
{
    ActivateWindow(myWindow);
    WindowToFront(myWindow);
    window_inactive = FALSE;

    return(TRUE);
}

BOOL wakeup(void)
{
    if (!(myWindow = Open_Window((SHORT) XORIGIN, (SHORT) YORIGIN,\
            (SHORT) XLENGTH, (SHORT) YLENGTH, NULL, FWINDOWFLAGS,\
            (UBYTE *)"Filer", (struct Screen *)0)))
    {
    /* Flash the screen if there is not enough memory to complete*/
    /* the operation. Wait for the operator to free up some memory.*/
        DisplayBeep(NULL);
    }else{
        sleeping = FALSE;
        myWindow->UserPort = myPort;
        ModifyIDCMP(myWindow, FWINDOWIDCMP);
        waitsigs = (PortSignal|EventSignal|RexxSignal);
        SetMenuStrip(myWindow, MenuList);
    }

    return(TRUE);
}

BOOL closeit(void)
{
    sleeping = TRUE;
    window_inactive = FALSE;
    if (myWindow->MenuStrip)
        ClearMenuStrip(myWindow);
    myWindow->UserPort = NULL;
    CloseWindow(myWindow);
    waitsigs = (EventSignal|RexxSignal);

    return(TRUE);
}

struct IntuiText b_txt = {0, 1, JAM2, 5, 5, NULL, NULL, NULL};
struct IntuiText p_txt = {0, 1, JAM2, 5, 3, NULL, (UBYTE *)"OK", NULL};

void inform(struct Window *myWindow)
{
    b_txt.IText = (UBYTE *)error_str;
    AutoRequest(myWindow,&b_txt,0L,&p_txt,0L,0L,\
        (long)(IntuiTextLength(&b_txt)+50L),50L);
}

BOOL install_input_handler(struct MsgPort *myPort)
{
    BOOL success;

    success = FALSE;

    /* allocate the memory for my input handler Interrupt structure */
    if (!(myHandler = (struct Interrupt *)AllocMem(sizeof(struct Interrupt),\
            MEMF_PUBLIC|MEMF_CLEAR)))
    {
            error_str = "Insufficient Memory - can't allocate Handler memory.";
            goto iih_exit;
    }

    /* allocate the memory for my standard IO requester structure */
    if (!(myRequest = (struct IOStdReq *)CreateExtIO(myPort,\
            sizeof(struct IOStdReq))))
    {
            error_str = "Insufficient Memory - can't allocate IORequest memory.";
            goto iih_exit;
    }

    /* open the input device for my use */
    if (OpenDevice("input.device", NULL, (struct IORequest *)myRequest, NULL))
    {
            error_str = "Error - Unable to open Input Device.";
            goto iih_exit;
    }
```

```c
    }

    success = TRUE;

    /* prepare the Interrupt structure to be used as an input handler */
    myHandler->is_Code = myEventhandler; /* routine entry point */
    myHandler->is_Data = NULL;                          /* I don't need a data area */
    myHandler->is_Node.ln_Pri = 51;        /* put handler right before Intuition */
    myHandler->is_Node.ln_Name = "Filer_Handler"; /* my handler name */

    myRequest->io_Data = (APTR)myHandler;  /* address of Interrupt structure */
    myRequest->io_Command = IND_ADDHANDLER;

    /* send the IO request to the input device to add my input event handler */
    DoIO((struct IORequest *)myRequest);

iih_exit:

    if (!success)
    { /* if this routine failed for any reason, I need to release */
      /* the memory I allocated. */
        if (myHandler)
        {
                FreeMem(myHandler, sizeof(struct Interrupt));
                myHandler = 0;
        }
        if (myRequest)
        {
                FreeMem(myRequest, sizeof(struct IOStdReq));
                myRequest = 0;
        }
    }

    return(success);
}

void close_input_handler(void)
{
    /* prepare to remove my input event handler */
    myRequest->io_Data = (APTR)myHandler; /* address of my Interrupt struct */
    myRequest->io_Command = IND_REMHANDLER;

    /* send the IO request to actually remove my input event handler */
    DoIO((struct IORequest *)myRequest);

    /* close the input device as far as my program is concerned */
    CloseDevice((struct IORequest *)myRequest);

    /* free my standard IO request structure */
    DeleteExtIO((struct IORequest *)myRequest);

    /* release my Interrupt structure memory */
    FreeMem(myHandler, sizeof(struct Interrupt));

    return;
}

BOOL dispatch(struct RexxMsg *myRexxMsg, BOOL (*fpt)(void),\
            char *argstr, struct RexxData *myRexxData)
{
    BOOL success;

    success_str = NULL; /* initialise success string to NULL */

    /* Call the handler designated by the rexxapp.library */
    success = (*fpt)();

    /* Setup the Result returns depending on error and success returns */
    if (success)
    {
            /* Success, both result values = 0. Return success_str */
            SetupResults( 0L, 0L, (STRPTR)success_str, myRexxMsg, myRexxData);
    }else{
            /* Error. Return error level 10 and a NULL result string */
            SetupResults(10L, 10L, NULL, myRexxMsg, myRexxData);
    }

    return(success);
}

void error_rtn(ULONG Code, char *Str, struct RexxMsg *rm, struct RexxData *rd)
{
    /* Very simple error processing routine. Simply display the error */
    /* string in the form of an auto requester to inform the user that an */
    /* error has occured */

    error_str = Str;
    inform(myWindow);

    return;
}

void process_rtn(ULONG Code, char *Str, struct RexxMsg *rm,\
            struct RexxData *rd)
{
    /* This is just a dummy routine as required by the rexxapp.library */
    /* I don't care for any return strings so just return immediately when */
    /* called and let rexxapp.library take care of the message deletion. */
```

```c
	return;
}

BOOL processmenu(USHORT selection)
{
    BOOL alldone;
    ULONG Menunum, Itemnum, Subnum;

    alldone = FALSE;

    while (selection != MENUNULL)
    {
        Menunum = MENUNUM(selection);
        Itemnum = ITEMNUM(selection);
        Subnum = SUBNUM(selection);
        error_str = NULL;

        switch (Menunum)
        {
            case 0:
                switch (Itemnum)
                {
                    case 0:
                        if (!newfile())
                            if (error_str)
                                inform(myWindow);
                        break;
                    case 1:
                        if (!openfile())
                            if (error_str)
                                inform(myWindow);
                        break;
                    case 2:
                        if (!savefile())
                            if (error_str)
                                inform(myWindow);
                        break;
                    case 3:
                        alldone = TRUE;
                        break;
                    default:
                        error_str = "Unexpected Item selection from Project menu.";
                        inform(myWindow);
                } /* Itemnum switch */
                break;
            default:
                error_str = "Unexpected Menu selection.";
                inform(myWindow);
                break;
        } /* Menunum switch */

        selection = ((struct MenuItem *)ItemAddress(MenuList,\
                        (LONG)selection))->NextSelect;
    } /* while */

    return(alldone);
}

BOOL newfile(void)
{
    BOOL success, definitioncomplete;
    UBYTE definitionstring[21];
    SHORT YPsn;
    const SHORT XPsn = 18;
    struct definition *newdef;
    struct Window *NFwindow;
    struct Gadget *Selectedgadget;
    struct IntuiMessage *NFintuimsg;
    struct IntuiText Definitiontext;
    struct {
        unsigned int typeselected : 1; /* bit-fields to tell me when all */
        unsigned int sizeselected : 1; /* three items have been selected */
        unsigned int nameselected : 1;
        unsigned int is_raw       : 1; /* is it a raw or key type) */
    } flags;

    success = definitioncomplete = FALSE;
    flags.typeselected = 0;
    flags.sizeselected = 0;
    flags.nameselected = 0;
    fieldcount = 0;
    YPsn = 16;
    Definitiontext.FrontPen = FCOL;
    Definitiontext.BackPen = BCOL;
    Definitiontext.DrawMode = JAM1;
    Definitiontext.LeftEdge = XPsn;
    Definitiontext.ITextFont = &MyFont;
    Definitiontext.IText = definitionstring;
    Definitiontext.NextText = NULL;

    if (fileptr) /* close any open file before creating a new one */
    {
        Close(fileptr);
        fileptr = NULL;
    }

    if (!set_up_deflist())
        goto nf_exit;

    if (!(NFwindow = Open_Window((SHORT)191, (SHORT)17, (SHORT)275,\
            (SHORT)164, NFWINDOWIDCMP, NFWINDOWFLAGS, (UBYTE *)"New File",\
            (struct Screen *)0)))
    {
        error_str = "Can't open New File window.";
        goto nf_exit;
    }

    /* Print the heading text for each gadget */
    PrintIText(NFwindow->RPort, &HeadingTextList, 0L, 0L);
    /* Add the Type, Size and Name gadgets */
    AddGList(NFwindow, &GadgetList, -1, -1, NULL);
    RefreshGList(&GadgetList, NFwindow, NULL, -1);

    while (!definitioncomplete)
    {
        WaitPort(NFwindow->UserPort);

        while (NFintuimsg = (struct IntuiMessage *)GetMsg(NFwindow->UserPort))
        {
            switch (NFintuimsg->Class)
            {
                case GADGETUP:
                    Selectedgadget = (struct Gadget *)NFintuimsg->IAddress;
                    switch (Selectedgadget->GadgetID)
                    {
                        case RAW:
                            flags.typeselected = 1;
                            flags.is_raw = 1;
                            break;
                        case KEY:
                            flags.typeselected = 1;
                            flags.is_raw = 0;
                            break;
                        case SIZE:
                            flags.sizeselected = 1;
                            break;
                        case NAME:
                            flags.nameselected = 1;
                            break;
                        default:
                            error_str = "Unexpected gadget type!";
                            inform(NFwindow);
                    } /* gadget ID switch */
                    if ((fieldcount < MAXFIELDS) && flags.typeselected &&\
                            flags.sizeselected && flags.nameselected)
                    {
                        if (!addlistitem((char *)NamegadgetSIBuff, flags.is_raw,\
                                (UBYTE) atoi(SizegadgetSIBuff)))
                        { /* failure indicates insufficient memory */
                            success = FALSE;
                            definitioncomplete = TRUE;
                        }else{
                            ++fieldcount;
                            if (flags.is_raw)
                            {
                                strcpy((char *)definitionstring, "RAW ");
                            }else{
                                strcpy((char *)definitionstring, "KEY ");
                            }
                            strcat((char *)definitionstring, (char *)SizegadgetSIBuff);
                            strcat((char *)definitionstring, " ");
                            strcat((char *)definitionstring, (char *)NamegadgetSIBuff);
                            Definitiontext.TopEdge = YPsn;
                            PrintIText(NFwindow->RPort, &Definitiontext, 0L, 0L);
                            YPsn += 10;
                            flags.typeselected = 0; /* reset selection indicators */
                            flags.sizeselected = 0;
                            flags.nameselected = 0;
                            RemoveGList(NFwindow, &GadgetList, -1);
                            SizegadgetSIBuff[0] = 0; /* clear the string gadget */
                            NamegadgetSIBuff[0] = 0; /* buffers */
                            AddGList(NFwindow, &GadgetList, -1, -1, NULL);
                            RefreshGList(&GadgetList, NFwindow, NULL, -1);
                        }
                    }
                    if (fieldcount == MAXFIELDS)
                    {
                        DisplayBeep(NULL);
                    }
                    break;
                case CLOSEWINDOW:
                    success = definitioncomplete = TRUE;
                    break;
                default:
                    error_str = "Unexpected Intuition class message.";
                    inform(NFwindow);
                    break;
            } /* class switch */
            ReplyMsg((struct Message *)NFintuimsg);
        }
    }

nf_exit:

    if (NFwindow)
        CloseWindow(NFwindow);

    return(success);
```

```c
}

BOOL set_up_deflist(void)
{
  BOOL success = FALSE;

  if (deflist)
  {
        freedeflist();
  }

  if (!(deflist = (struct List *)AllocMem((long)sizeof(struct List),\
    MEMF_CLEAR)))
  {
        error_str = "No memory for record definition list.";
        goto sud_exit;
  }

  NewList(deflist); /* Initialize list header */
  success = TRUE;

sud_exit:
  return(success);
}

BOOL addlistitem(char *name, unsigned int is_raw, UBYTE size)
{
  struct definition *newdefinition;
  BOOL success;

  success = FALSE;

  if (!(newdefinition = (struct definition *)AllocMem((long)\
        sizeof(struct definition), MEMF_CLEAR)))
  {
        error_str = "No memory for additional fields.";
        goto ai_exit;
  }

  strcpy(newdefinition->Name, name);
  newdefinition->defNode.ln_Name = newdefinition->Name;
  newdefinition->type = is_raw;
  newdefinition->size = size;

  AddTail(deflist, (struct Node *)newdefinition);

  success = TRUE;

ai_exit:

  return(success);
}

void freedeflist(void)
{
  struct definition *currentnode, *nextnode;

  /* initialize first entry ; look ahead to next entry ; change entries */
  for ( currentnode = (struct definition *)(deflist->lh_Head) ;\
        nextnode = (struct definition *)(currentnode->defNode.ln_Succ) ;\
        currentnode = nextnode )
  {
    FreeMem(currentnode, sizeof(struct definition)); /* free each entry */
  }

  FreeMem(deflist, sizeof(struct List));
  deflist = NULL;
  fieldcount = 0;

  return;
}

BOOL openfile(void)
{
  int loopcount;
  BOOL success = FALSE;
  struct definition tempdef;

  if (!getafilename((UBYTE *)"Load File"))
    goto of_exit;

  if (!set_up_deflist())
    goto of_exit;

  if (fileptr)
  {
    Close(fileptr);
    fileptr = NULL;
  }

  if (!(fileptr = Open(FileName, MODE_OLDFILE)))
  {
    error_str = "Unable to open requested file.";
    goto of_exit;
  }
```

```c
  /* read in the number of definitions in record make-up */
  if (!Read(fileptr, (char *)&fieldcount, sizeof(int)))
  {
    error_str = "Trouble reading in field count from file.";
    goto of_exit; /* problems with reading from file */
  }

  /* read in all the definitions */
  for (loopcount = 0; loopcount < fieldcount; loopcount++)
  {
    if ((!Read(fileptr, (char *)&tempdef, sizeof(struct definition))) ||
        (!addlistitem(tempdef.Name, tempdef.type, tempdef.size)))
    {
      error_str = "Trouble establishing field definitions.";
      Close(fileptr);
      fileptr = NULL;
      freedeflist();
      goto of_exit;
    }
  }

  success = TRUE;

of_exit:

  return(success);
}

BOOL savefile(void)
{
  BOOL success = FALSE;
  struct definition *currentnode, *nextnode;

  if (!getafilename((UBYTE *)"Save File"))
    goto sf_exit;

  if (FileName[0])
  {
    if (fileptr)
    {
        Close(fileptr); /* close any open file first */
        fileptr = NULL;
    }

    if (!(fileptr = Open(FileName, MODE_NEWFILE)))
    {
        error_str = "Unable to create a new file.";
        goto sf_exit;
    }

    /* write the number of definitions in record make-up */
    if (!Write(fileptr, (char *)&fieldcount, sizeof(int)))
    {
        error_str = "Unable to write field count to file.";
        goto sf_exit; /* problems with writiing to the file */
    }

    /* write field definitions */
    /* initialize first entry ; look ahead to next entry ; change entries */
    for ( currentnode = (struct definition *)(deflist->lh_Head) ;\
          nextnode = (struct definition *)(currentnode->defNode.ln_Succ) ;\
          currentnode = nextnode )
    {
        if (!Write(fileptr, (char *)currentnode, sizeof(struct definition)))
        {
          error_str = "Unable to write all field definitions to file.";
          goto sf_exit;
        }
    }

    success = TRUE;
  }

sf_exit:
  return(success);
}

BOOL quitprog(void)
{
  /* This is all that is needed to indicate that the program is done */
  all_done = TRUE;
  return(TRUE);
}

BOOL getafilename(UBYTE *RequesterText)
{
  BOOL success = FALSE;
  USHORT activeflag;

  FileName[0] = (char) 0; /* clear any previous file name */
  FNGSIBuff[0] = (UBYTE) 0; /* clear the requester string buffer */
  myReq.ReqText->IText = RequesterText;

  SizeWindow(myWindow, (SHORT) XDELTA, (SHORT) YDELTA);

  Delay(50); /* allow window to resize befor continuing */

  if (!Request(&myReq, myWindow))
  {
```

```c
        SizeWindow(myWindow, (SHORT) -XDELTA, (SHORT) -YDELTA);
        error_str = "Unable to open a filename requester.";
        goto gfn_exit;
    }

    /* pause while requester is active */
    /* initialize test variable ; test value ; update test variable */
    for ( activeflag = myReq.Flags & REQACTIVE ;\
              activeflag ;\
              activeflag = myReq.Flags & REQACTIVE )
    { /* slow down the loop, don't hog all the microprocessor time */
      Delay(50);
    }

    SizeWindow(myWindow, (SHORT) -XDELTA, (SHORT) -YDELTA);

    if (FNGSIBuff[0] != 0) /* non zero indicates user made a file selection */
    {
      strcpy(FileName, (char *)FNGSIBuff); /* copy the name into our buffer */
      success = TRUE;
    }

gfn_exit:
    return(success);
}

#asm
        ; .... ENTRY VALUES ....
        ; a0 = pointer to the event linked list
        ; a1 = pointer to my data segment (NULL)
        ; .... RETURNS ....
        ; d0 = return the new event linked list here

        xdef _myEventhandler      ;make entry point available to C code
        xref _myTask              ;pointer to Filer Task structure
        xref _EventSignal         ;pointer to Filer Wake-up event signal

        ; some equates that may event handler needs
```

```asm
ie_NextEvent     equ 0
ie_Class         equ 4
ie_Code                      equ 6
ie_Qualifier     equ 8
IECLASS_NULL     equ $00
IECLASS_RAWKEY   equ $01
IEQUALIFIERB_CONTROL         equ $03
IEQUALIFIERB_LALT            equ $04
IEQUALIFIERB_RALT            equ $05

_myEventhandler:
            move.l  a0,-(sp)    ;save the pointer to the first event structure
check:      cmp.b   #IECLASS_RAWKEY,ie_Class(a0) ;rawkey event?
            bne.s   next    ;no, so go to the next event
            cmp.w   #$27,ie_Code(a0)    ;'f' key depressed?
            bne.s   next    ;no, don't need this event
            move.w  ie_Qualifier(a0),d0 ;move qualifier into d0
            btst    #IEQUALIFIERB_CONTROL,d0 ;is CONTROL down?
            beq.s   next    ;bit not set, don't need this event
            btst    #IEQUALIFIERB_LALT,d0 ;is left ALT down?
            bne.s   ok      ;bit set, I want this event
            btst    #IEQUALIFIERB_RALT,d0 ;is right ALT down?
            beq.s   next    ;bit not set, don't need this event
ok:         move.l  _SysBase#,a6
            move.l  a0,-(sp)                    ;save event list pointer
            move.l  _myTask,a1      ;put Task structure pointer in a1
            move.l  _EventSignal,d0 ;put wake-up signal in d0
            jsr     -324(a6)        ;Signal(myTask, EventSignal)
            move.l  (sp)+,a0                    ;restore event list pointer
            move.b  #IECLASS_NULL,ie_Class(a0)  ;remove event
next:       move.l  ie_NextEvent(a0),d0 ;get next event
            move.l  d0,a0   ;put event structure address in a0
            bne.s   check   ;go back if there is another event
            move.l  (sp)+,d0 ;restore the event list
            rts             ;return from handler

#endasm
```

# The Amiga and the
# MIDI
# Hardware
# Specification

A revolution began in 1983 when several electronic musical instrument manufacturers joined to publish a standard describing a system of communication between music synthesizers. The Musical Instrument Digital Interface or MIDI is now available on almost all electronic musical instruments. MIDI allows synthesizers, sequencers, rhythm machines, and effects boxes to control one another and has greatly expanded the role computers now play in music. With right software, an Amiga connected to a MIDI instrument can record and play back a performance, edit and print a musical score, act as a synthesizer patch editor and librarian, and even teach you how to play!

MIDI is a digital serial communication specification that consists of a physical standard which describes the type of connector and cable to interconnect MIDI devices, a hardware standard that specifies the design of the electrical interface and a software standard which establishes the type and format of MIDI messages. MIDI has become such an important communication system that some models of the Macintosh and Atari have MIDI ports built-in.

While this is not intended to be a do-it-yourself construction guide to building a MIDI interface, with some degree of technical skill and the information in this article, you'll find it quite easy to put one together. Be aware that you bear complete responsiblity for any damage caused by the design and construction of such an interface based on information presented in this article. Unlike an annoying program bug, an error in your design could cause severe damage to your computer and any connected MIDI device. Repair technicians are notably unsympathetic about hardware hackers' mistakes and are apt to charge accordingly. It goes without saying that the addition of any home brew circuit effectively voids the manufacturer's warranty. Proceed cautiously!

## by James Cook

## Learn the ins and outs of the mysterious MIDI Hardware Specification... and build your own MIDI Interface!

### The Physical Specification

The typical MIDI compatible device usually has three 5-pin female DIN ports. The MIDI IN port receives transmissions from other devices, the MIDI OUT port transmits data to other devices and the MIDI THRU port acts as a MIDI OUT for all data received at the MIDI IN port. Each MIDI OUT port can supply one and only one MIDI IN port. For this reason some MIDI devices have several MIDI OUT ports to transmit signals to multiple MIDI instruments. A MIDI cable consists of a shielded single twisted pair of wires with male DIN plugs on each end. The shield of the cable is connected to pin 2 and the twisted pair is connected to pins 4 and 5. Pins 1 and 3 are not defined and the specification recommends they not be used. The cable may not be longer than 50 feet. Note that the 5-pin DIN cables sold to interconnect some types of hi-fi equipment are not MIDI cables. Hi-fi DIN cables do not have the shield connected to pin 2 and therefore these cables are more susceptible to induced electrical noise interfering with the MIDI signal.

### The Software Specification

While this article is primarily a discussion of the MIDI hardware standard, some mention of the software specification is needed. A complete treatment of MIDI software programming, however, would require several articles. Briefly, most MIDI communications consist of multi-byte messages which are constructed of one Status byte followed by one or two Data bytes. Messages are divided into two types, Channel and System. Channel messages include a four-bit number in the least significant nibble of the Status byte to indicate which one of 16 MIDI devices is to respond to the message. The most significant nibble signals the receiver as to the function the transmitter expects it to perform. The most significant bit in the Status byte is always set to differentiate Status bytes from Data bytes. Two exceptions to this message format are System Real-Time and System Exclusive messages. Real-Time messages are used for synchronizing all MIDI devices in the system including sequencers and rhythm units. Exclusive messages are special communications which any manufacturer may define for use between its own MIDI devices.

For example, to send a message to MIDI device 7 to play middle C mezzo-piano, the transmitter would send 10010100 00111100 01000000. 1001 = Note On, 0100 = channel 7, 00111100 = 60 (middle C is note number 60), 01000000 = 64 (a note velocity roughly equal to striking a piano key mezzo-piano). Note that this message simply commands device seven to begin playing middle C. Middle C will continue playing until it is commanded to stop with a Note Off message or a Note On message with zero velocity. Another important message is the Program Change message which commands a receiving device to select a new voice or patch setting. There are several other messages as well including control signals from pitch wheels, breath controllers, etc. MIDI software has greatly expanded from that defined in the original specification. The members of the International MIDI Association recognized that this was inevitable and desirable and actively encourages member manufacturers to make readily available information on new commands or message formats that a firm may introduce on new equipment.

### The Hardware Specification

The MIDI signal itself is a 5 mA current loop operating at a speed of 31.25 Kbaud (+/- 1%), asynchronous. Each byte is transmitted by a Universal Asynchronous Receiver/Transmitter or UART integrated circuit and is preceded by a start bit and followed by a stop bit. While the transmitter design is relatively noncritical as long as the 5 mA current loop specification is adhered to, the receiver circuit must contain an optoisolator which has a rise and fall time less than two microseconds. The main purpose of the optoisolator is to ensure that the transmitting MIDI device is electrically isolated from the MIDI receiver. An optoisolator with as fast a rise and fall time as possible helps reduce the dreaded "MIDI delay" problem which occurs when chaining more than three or four instruments. MIDI delay will cause the farther instruments to play noticeably after a key is pressed on the transmitting instrument.

The MIDI specification identifies two optoisolators, the Sharp PC-900 and the HP 6N138, as acceptable although others may be satisfactory. The hardware specification includes a schematic diagram of the final receiver, transmitter and thru portions of a MIDI interface. See Figure 1.

The MIDI OUT port has pin 4 of the female DIN jack connected to +5 volts through a 220 ohm resistor. Pin 5 is the transmitted signal which also passes through a 220 ohm resistor. One or more invertors, as shown in Figure 2, are used to buffer the output of the UART and to ensure that current is ON for a logical 0. The MIDI IN port receives the signal at pin 4 which is passed to the input of an optoisolator through yet another 220 ohm resistor. Inside the optoisolator is an LED which turns on and off with the received signal. The 0 to 5 volt signal transmitted from the UART divided by the total resistance of the three 220 ohm resistors and the LED makes up the 5 mA signal. The light emitted by the LED inside the optoisolator falls upon a photoreceiver which switches a +5 volt signal on and off. This signal is passed on to the receive section of a serial UART and is also buffered and connected to the MIDI THRU port, if present, in the same way as a MIDI OUT port. Pin 2 is left unconnected at the MIDI IN port to prevent shield ground loops which could induce electrical interference into the MIDI signal.

## The Amiga Interface

Unlike Apple and Atari, Commodore did not design a MIDI port into the Amiga, choosing to rely on third party developers to provide a suitable interface. Fortunately, the Amiga engineers did provide a programmable UART integrated circuit for serial communications capable of being set to communicate at the 31.25 Kbaud speed required by MIDI. Unlike most of the serial ports available for PC clones, this allows the Amiga to send and receive serial communications at the rather speedy rate demanded by MIDI standards without the addition of a suitable UART in the MIDI interface.

The only thing an Amiga MIDI interface must provide in addition to the standard MIDI circuit is the hardware to properly implement the 5 mA current loop output and the 0 to 5 volt input from the optoisolator. Since the Amiga's serial port is a standard RS-232 port normally used to communicate with modems and serial printers, it is designed to send and receive signals which switch between -12 and +12 volts. Some form of signal level conversion is necessary to convert the +/- 12 volt signal to 0 to +5 volts. The two most common level conversion chips are the 1488 and 1489. The 1489 accepts almost any voltage level input and converts it to 0 to +5 volts. The 1488 performs the opposite function, converting a 0 to +5 volt signal to a signal which switches between voltage levels



**Figure One**

MIDI Standard Hardware
Based on MIDI 1.0 Specifications



**Figure Two**

Amiga Interface to
MIDI Hardware Specification

applied at pins 1 and 14. These two chips are also used inside the Amiga itself to convert the 0 to +5 volt signals generated by the internal logic to drive the RS-232 port. See Figure 2.

To convert a received MIDI signal, one must feed the output of the optoisolator directly to pin 2 of a 1488. With the exception of the Amiga 1000, all Amiga serial ports have +12 volts connected to pin 9 and -12 volts connected to pin 10 of the DB-25

connector. The 1488 can be powered by connecting pin 9 of the DB-25 connector to pin 14 of the IC and connecting pin 10 of the serial connector to pin 1 of the chip. Pin 7 (Ground) should be connected to pin 7 of the DB-25 connector. Pin 3 of the 1488 will provide a +/- 12 volt signal to the receive input, pin 3, of the serial port. While the 1488 contains four converting gates, only one is needed. Amiga 1000 owners will find this method of signal conversion difficult to implement since the 1000 does not supply -12 volts through the serial connector (+12 volts is available from pin 23). Amiga 1000 users will either have to supply -12 volts externally or use an integrated circuit like the Sipex SP232 which contains a voltage doubler and invertor circuit to convert +5 volts to a +/- 10 volt signal level, which will be adequate to drive the RS-232 port.

The transmitted signal from pin 2 of the Amiga's serial port must be converted from +/- 12 volts to a 0 to +5 volt signal. Since only the Amiga 1000 provides a +5 volt power output from its DB-25 connector (pin 21), a 7805 +5 volt regulator should be used. Power from pin 9 of the DB-25 connector is applied to pin 1 of the 7805 and ground from pin 7 of the connector is wired to pin 3. The +5 volt output from pin 2 of the regulator is connected to pin 14 of the 1489 IC. The 1489, like the 1488, also provides four gates. This allows the use of one 1489 to supply three MIDI OUT ports and one MIDI THRU port. Connect pin 2 of the Amiga to pins 1, 4, and 10 of the 1489 and you'll have MIDI OUT's on pins 3, 6 and 8. Simply connect pin 3 of the 1488 to pin 13 of the 1489 and pin 11 will provide a MIDI THRU output. Since the Sipex SP232 chip contains two RS-232 to digital receivers and two digital to RS-232 transmitters, you may want to consider using it if you need a only single MIDI OUT, a MIDI THRU and a MIDI IN port. The advantage to using this chip is that you could provide an external source of +5 volts to power the circuit, and the interface could be connected to any Amiga computer (with a gender changer for the 1000, of course). Small wall socket power supplies of +5 volts are readily available and relatively inexpensive.

### Construction Hints

Construction of an Amiga MIDI interface can be quite simple. Assembly techniques are non-critical and most of the electronic components required are available at your local Radio Shack or other electronic parts outlet. Only the optoisolator or the Sipex chip will have to be special ordered. Total cost will be around $30.

Here are a few tips which can be applied to any hardware you intend to connect to your computer to reduce the chance of accidentally damaging your machine. All designs should be sketched and thoroughly researched before anything is assembled. Your initial circuit should be built on a solderless breadboard which allows quick and easy changes. After double checking your breadboarded circuit for accuracy, connect your circuit to the Amiga in the safest way possible. Since the Amiga's power supplies are reasonably well protected against shorts or overloads, simply powering up the circuit should be your first step. For the MIDI interface, for example, connect only the +/- 12 volt pins and ground to your prototype circuit from the computer *with the power off*. With a voltmeter connected to ground and +12 volts, switch on the computer. If the voltage does not immediately rise

to around +12 volts, *switch off the computer*. Disconnect the interface and switch the computer on again. Check between pin 7 (ground) and pin 9 (+12 volts) on the RS-232 connector to make sure you haven't damaged the +12 volt supply. If you are powering your interface through the serial port of your computer, make sure you have a serial cable with all 25 pins connected. Most serial cables have only the most commonly used pins connected. If you are trying to use one of these cables, the +/-12 volt lines will not be connected to your interface and the circuit will not work. Be careful using a serial cable with all 25 pins connected with any other serial device. Some modems, serial printers, etc., may be damaged by the presence of these 12 volt signals.

Once you've established that the circuit is not shorting or overloading the +/-12 volt supplies, connect the transmit and receive pins to the interface and again measure the voltage after switching on the computer. If everything seems to be working properly, switch off the computer and plug in a MIDI cable between the MIDI OUT port and MIDI IN port of your interface. Check your voltage levels once again. Now boot up your machine and run your favorite telecommunications or terminal program. It doesn't really matter what baud rate or bit settings are used, but make sure your local echo is turned off. Now, type on the computer keyboard. If everything is working properly the telecommunication or terminal program will be sending the characters you type out the serial port and into your interface. The interface will transmit the characters using the MIDI signal standard from the MIDI OUT port and receive the same characters back on the MIDI IN port. The interface will convert the received signals back to the RS-232 standard and the typed characters will be echoed on the screen. Just to make sure you've set your communications program up right, disconnect the MIDI cable and the characters should no longer be echoed to the screen.

### Using the Interface

You're now ready to give the interface a try! This interface will work with almost all MIDI software available for the Amiga. It has been tested with *Deluxe Music Construction Set, Bars & Pipes, Tiger Cub* and several public domain programs. Plug a MIDI cable from the MIDI OUT port of your interface to the MIDI IN port of any MIDI compatible keyboard. Use another MIDI cable to connect between the MIDI OUT port of the keyboard to the MIDI IN of your computer. The only time you'll need the MIDI THRU port is when you want another MIDI device to receive the same signals received by the computer.

Follow the instructions on accessing a MIDI device carefully when running a MIDI program. Deluxe Music Construction Set, for example, requires you to explicitly enable the MIDI input and output features from the menu. You also have to declare a MIDI channel and 'program change' or voice setting in your score. While DMCS is capable of providing an excellent means of directly editing a score into a format to drive MIDI devices, it does not have the performance input features of a sequencer program like Bars & Pipes, *Music-X* or Tiger Cub. If your MIDI keyboard is a synthesizer, you may wish to look into the many patch librarians available to help organize and store your settings.

☑

# Programming the Amiga in Assembly Language
## Part I—Getting Started

by William P. Nee

## INTRODUCTION

Writing assembly language programs for the Amiga is not that complicated. The commands are short, simple, and varied enough to let you do two or three routines with just one phrase. I have placed a glossary of commands at the end of this article, and also I will discuss each one when it's used. In this series I'll cover several types of programming techniques and demonstrate a new procedure in each article. All of the machine language programs may be assembled using the A68K Assembler and Blink, both excellent public domain programs, included on this disk; no other files or "includes" are necessary. You will need to refer to previous articles as we go along since not much old information will be repeated. All of the programs will run on an Amiga 500 using WorkBench v1.2 or higher. Thanks to Adrian Kotik for debugging, proof-reading, and putting these articles together.

First, let's get some terminology out of the way. Your computer is filled with memory locations called bytes. Each byte holds eight bits (called, from left to right, bit7 to bit0) of information either a "0" or a "1." Using just these two numbers (called binary or Base 2 system) all values, commands, and instructions can be represented. In one byte alone, there are 256 ($2^8$) possible combinations of "0" and "1." If you combine two bytes you get one "word," and two words make up one "long word" that holds 32 bits of information. With only 512,000 bytes available in a standard Amiga 500 with no additional memory, you can begin to get some idea of how much storage room is available.

Even though the computer uses the Base 2 for its numbers, programmers find it easier to use the decimal Base 10 or the hexadecimal Base 16. With the latter, you have to add six additional numbers (called "A" through "F") to the usual 1 through 9. So 10 in the Base 16 is 9+1 or A, 11 is B, and so forth; $16_{10}$ is $10_{16}$. Since the largest number a byte can hold is $255_{10}$, it is often convenient to think of that as $FF_{16}$. To distinguish numbers in the Base 16—usually called HEX—from numbers in the Base 10, the HEX numbers are prefixed with #$ and HEX locations in the computer's memory are prefixed with $.

## REVIEWING THE BASES

If you're familiar with Base 2 math, you might want to skip this section. With just two numbers to use, the binary system isn't that complicated but this is a good time to review it. Addition is simple; 0+1=1 and 1+1=10. Since each digit represents a power of two, the number $111_2$ would represent $(1*22)+(1*21)+(1*20)$ or $7_{10}$. As numbers get higher, they also get longer to write so you can see why the Base 16 or HEX became popular. Now with each number representing a power of 16, the number #$FF would mean $(F*161)+(F*160)$ or 255 - remember that A through F in HEX actually represent 10 through 15 in Base 10.

The most confusing part of computer math is probably negative numbers. Since there is room for only those 0's and 1's, how do we get a minus sign in there? You have to think of the numbers you're using as being on a giant wheel going from 0 to, let's say, #$FFFF. This number fills all the bits of one word so it makes a convenient example. Turning our imaginary wheel to the left will show all the positive numbers: 0,1,2, etc. Rotating in the opposite direction must then indicate the negative numbers. Since the first negative number is #$FFFF, it must be the same as -1; next would be #$FFFE and it must be -2 and so forth. Half way through the wheel in either direction is the boundary between positive and negative numbers.

The only difference between our positive and negative numbers is that the left-most bit (or Most Significant Bit) of any negative number must always be set, that is, be a "1"; the MSB of any positive number must always be a "0." This will be true for whatever size number you use—byte, word, or long word. Since we said that half-way through the wheel was the change-over between plus and minus, we can effectively use only half of our number range if we are going to need negative values. That is, if your numbers range from 0 to #$FF, the portion from 0 to #$7F is positive and from #$FF to #$80 is negative. So the smallest number is -128 and the largest is +127. If you need numbers outside these values you'll have to increase the size of the range, probably from one byte to one word or #$FFFF. Now the numbers from 0 to #$7FFF are positive and from #$FFFF to #$8000 are negative.

I said that this was necessary if you wanted to use negative numbers. But what if all your values will be positive ("0" is always considered positive)? No problem; use the full range of all your numbers, that is, from 0 to #$FF (0 to 255) or from 0 to #$FFFF. The computer will work the same, but is it up to you to interpret the results.

When we get to BRANCH commands later on, you'll see that some of the branches will depend on whether you could have a negative or positive result. More mistakes are probably made in this area than anywhere else. Your program can go along perfectly and then suddenly display the "Big Bang" theory simply

because you forgot that there could be negative values and didn't branch accordingly. My early examples will all use positive values so we won't have to worry about negative branching yet. And what about fractional numbers? That's even more fun!

## SETTING UP YOUR ASSEMBLER DISK

Enough theory. Let's actually get set up and try a simple program. I suggest you use a stripped down version of WorkBench adding A68K and Blink which are both included on this disk. There may be later versions of those two PD programs, but I've had no problems using these. You can create your own disk by copying a WorkBench disk and removing the unnecessary files or start with a blank disk, format it, use INSTALL to make it self-booting and then add these files. Name this disk ASSEMBLER:. I find it easier to put the programs and commands I'll be using in RAM: and RAM:C so my disk and start-up sequence look like:

| DIRECTORY OF ASSEMBLER | | |
|---|---|---|
| C DIRECTORY | L DIRECTORY | DEVS DIRECTORY |
| a68k | Port-Handler | narrator.device |
| blink | Ram-Handler | parallel.device |
| cd | ————— | printer.device |
| copy | S DIRECTORY | system-configuration |
| delete | startup-sequence | PRINTERS DIRECTORY |
| dir | ————— | (your printer) |
| ed | LIBS DIRECTORY | ————— |
| makedir | mathieeedoubbas.library | FONTS DIRECTORY |
| path | mathtrans.library | topaz.font |
| more | translator.library | TOPAZ.DIRECTORY |
| | | 11 |

There may be other C commands you want to add such as "type" or "info" but they probably won't be needed in RAM:C. Now that you have the necessary files on your ASSEMBLER disk, I suggest that you use the following startup-sequence.

| ASSEMBLER:S/STARTUP-SEQUENCE |
|---|

```
dir ram:
makedir ram:c
copy c/a68k|blink|cd|copy ram:c
copy c/delete|dir|ed|makedir ram:c
copy c/more ram:c
path ram:c
cd ram:
```

This startup-sequence will copy all of the files you will be using into RAM:C and will put you in the root directory of RAM: where you'll be saving and assembling your programs. At this point I suggest formatting a fresh disk and calling it PROGRAMS. Once the programs have been debugged and assembled in RAM:, the executable programs and source codes will be copied onto this disk. While all of the libraries on your ASSEMBLER disk won't be used immediately, you might as well start tracking them down now. They'll all be needed eventually.

I use ED to type the programs, but you may use any word processor that saves files in the ASCII mode. When writing a machine language program the routine heading goes immediately to the left; commands must be at least one space over but I use one tab over; the rest of the line, if any, must be at least one space or tab over from the command. Comments may be added but must be preceded by a semi-colon. It is always a good idea to add as may comments as possible; two months after you write a program it will be hard to remember what all those lines were supposed to do. And try to make routine headings a good understandable name. It's easier to figure out what "SendToPrinter" will do rather than "L10".

## YOUR FIRST ASSEMBLY LANGUAGE PROGRAM

This is a short, simple program that will get you started and let you check out your new disk. It causes the red power light on your Amiga to blink off and on six times. Right now don't worry about the commands and all those other cryptic symbols (we'll get to them later); just type it in as written. Boot up your Amiga with the ASSEMBLER disk and type in the following program using ED or your favorite word processing program. Since machine language programs use simple words and phrases I prefer using ED. It's also a fairly short program itself and takes up very little room in RAM:C.

```
start:
    move.l    sp,stack       ;save the Stack Pointer
number_of_times:
    moveq     #6,d1          ;number of times to blink
power_off:
    or.b      #2,$bfe001     ;set bit one to 1
    moveq     #0,d0          ;clear d0
delay1:
    subq.w    #1,d0          ;subtract 1 from d0
    bne.s     delay1         ;branch if not back to 0
delay2:
    subq.w    #1,d0
    bne.s     delay2
power_on:
    andi.b    #253,$bfe001   ;clear bit one to 0
    moveq     #0,d0
delay3:
    subq.w    #1,d0
    bne.s     delay3
delay4:
    subq.w    #1,d0
    bne.s     delay4
    subq      #1,d1          ;decrease number of blinks
    bne.s     power_off      ;branch if not finished
    move.l    stack,sp       ;restore the Stack Pointer
    rts                      ;return to CLI
    even                     ;force an even address
stack dc.l 0                 ;stack storage
    end                      ;end of listing
```

After typing this program, save the source code in RAM: as POWER.ASM; all A68K programs must end with .ASM. To assemble it, type A68K POWER.ASM. Since you're in RAM:, the disk won't have to come on and the assembly is extremely fast. If there are any mistakes, the assembly listing will stop and tell you what line has an error. If you were lucky enough not have to have

any errors the first time, try changing the first "move.l" to read "mave.l". Now when you assemble, you'll get an error in line #2 and the phrase "No Such Op-Code." Generally the A68K assembler will only catch syntax errors and some type mismatches (bytes, words, etc.). The assembler can't tell you if your program won't work properly or may crash.

After assembling a program, A68K creates a new file—in this case called POWER.O. This Object file must be linked to make it an executable program, so type BLINK POWER.O. Now you have another listing called POWER; this is the assembled program. At some later time you may want to delete the accumulated .O files. Before trying the assembled program, I would save it along with the source code to disk; if your program does crash, you'll loose everything in RAM:. You can either save your programs to the assembly disk or, as I usually do, to a second program-only disk called PROGRAMS. From RAM: type COPY POWER:ASM PROGRAMS: and COPY POWER PROGRAMS:. As you get more programs, you may want to create directories on your program disk and save programs in them, but for the time being you can save this short program directly to disk.

Now you're ready to try your program. Type POWER and the Amiga red light should blink six times and then stay on. Congratulations! For many of you that was probably your first assembly language program. By the way, throughout this series I use "assembly language" and "machine language" interchangeably. With your first program out of the way, it's time to get down to basics and discuss how the Amiga processes the machine language programs.

## GETTING TECHNICAL

The Amiga stores information in registers. There are eight 32-bit registers used for data (d0 to d7) and seven used for addresses (a0 to a6); an eighth register (a7) is available but it is generally used as a Stack Pointer (SP) for saving information locations. Part of yet another register (CC) tells you the "Condition Code" for any operation—mainly if an operation produced a value of 0 or not, and if the result was a plus or minus number. There are four main points to remember when writing Amiga machine language programs:

1. All main headings (library, array locations, etc.) will be at different locations when you first power-up; well, all but one. This forces you to write "locatable" programs but more about that in a later article.
2. Most subroutines are located in various libraries and are accessed by going to an "offset" location from the library.
3. Important items are structured; when setting up a screen, for example, there are various elements that must be defined in a specific order. This allows you to change parameters since they are at a fixed distance from the main item.
4. All address locations must be even addresses—this is usually handled by the assembler.

Finally, a brief discussion of the two major commands used in assembly language. MOVE is used to put a value or the contents of one register into another register. Some of the possible variations on MOVE are:

| | |
|---|---|
| MOVE.B | moves the right-most byte |
| MOVE.W | moves the right-most word |
| MOVE.L | moves the entire long word |
| MOVEA | moves only to an address register |
| MOVEQ | moves a signed byte value (a number from -128 to +127) to a data register |
| MOVEM | moves several registers at once |

The other command is LEA—Load Effective Address. It's used to move labeled locations to an address register or to increase the contents of an address register. Some examples are:

| | |
|---|---|
| LEA name,a1 | moves the location or address of "name" into a1 |
| LEA 100(a1) | increases the address in a1 by 100 |

Any command with a parenthesis around a register means "the contents of."

## HOW WE DID THAT?

With that general information, let's review how you caused the power light to blink several times. The MOVE.L command saves the contents of the Stack Pointer in a location called "stack." Next, a MOVEQ was used to store #6 in register d1. In the Amiga 500, location $BFE001 controls, among other things, the power light. If bit 1, the second bit from the right is "0" so the light stays on, but if that bit is set to 1, the light will go out. The Basic AND and OR commands have the same meaning in assembly language and are often used to force a bit location to 0 or 1. Any binary number OR 1 will always be 1 while any number OR 0 remains the same; any binary number AND 0 will always be 0 and any number AND 1 remains the same. So when you OR a byte with #2 ($10_2$) bit 1 is set to 1.

Next, register d0 is cleared with a MOVEQ #0 command — by the way, this is the quickest way to clear any data register. Then #1 is SUBtracted from d0 (resulting in #$FFFF); SUBQ and ADDQ may only be used with values between #1 and #8. The BNE (Branch if Not Equal to 0) will go back to "delay1:" until d0 finally reaches 0; the S indicates a Short branch (-128 to +127 bytes away). Then the entire delay is repeated so you can notice it. The light is turned back on by clearing bit 1 with ANDI.B #253 ($11111101_2$); the "I" means that an Immediate number will be used with the AND command. After two more delays register d1 is decreased by 1 and the entire sequence repeated. When d1 finally reaches 0, the original contents of the Stack Pointer are restored. The RTS (ReTurn from Subroutine) takes us back to CLI. EVEN is a code to align the assembler to an even address; as I mentioned earlier all addresses must begin at an even numbered location. After the actual program a space is reserved as "stack" using the storage command DC.L 0 (Define Constant as a Long word value of 0). Finally, END signifies the end of the listing.

Let's talk about those libraries in more detail. The major ones we'll use in this series are EXEC (tasks, memory, and ports), DOS (disk functions, read, write), GFX (graphics), MATH, MATHTRANS (sign, cosine, etc.), INT (windows and screens), and TRANSLATE (speech). While there are several more libraries, we'll work mainly with these. All the subroutines we use are

# AC's TECH Back Issues

# AC's TECH Disk
## *Volume 2, Number 1*

### A few notes before you dive into the disk!

• You need a working knowledge of the AmigaDOS CLI as most of the files on the
  AC's TECH disk are only accessible from the CLI.

• In order to fit as much information as possible on the AC's TECH Disk, we archived
  many of the files, using the freely redistributable archive utility 'lharc' (which is
  provided in the C: directory). lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  *Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.*

---

AC's TECH DISK
GOES HERE!

Please notify your
retailer if the
AC's TECH Disk
is missing.

*Be Sure to
Make a
Backup!*

We pride ourselves in the quality of our print
and magnetic media publications. However, in
the highly unlikely event of a faulty or dam-
aged disk, please return the disk to *PiM
Publications, Inc.* for a free replacement.
Please return the disk to:

*AC's TECH*
Disk Replacement
P.O. Box 869
Fall River, MA 02720-0869

located at a specific offset away from its library location. The PSET routine is, for example, 324 bytes away from the GFX library location; or, more correctly, 324 bytes away will tell you to jump to some location that actually contains the PSET routine. In addition to calling the routines, certain registers must contain specific information for the routine to work. As we use each library and routine, I'll include the offsets and required information.

As I had mentioned earlier, you won't know library addresses at power-up except for the EXEC library. It's address is always stored in location $4 and it's always available. The routine to open another library is called "OpenLibrary" and is located - 552 bytes away from the EXEC library. For this routine to work,

register a1 must contain the location of the library name you want to open, and d0 must contain the latest acceptable version number of that library, usually 0.

## OPENING A LIBRARY

Let's try opening the DOS library. This routine will work for all libraries by just changing the library name. The three main portions of this program are the offsets we'll use, the program itself, and a data/storage location. After the library is opened, we'll use two DOS routines to print a message and print the actual library location as a HEX number. The "Output" routine gets the CLI screen-handler and returns its location in d0. The "Write" routine will print a message or any characters in a string buffer;

## A68K ASSEMBLER COMMANDS

| OPCODES | SOURCE,DESTINATION |
|---------|--------------------|
| ADD.B/W/L | add a number or register to a register |
| ADDA | destination is address register |
| ADDI | immediate number |
| ADDQ | #1 through #8 |
| BCC | same as Branch if Higher or Same |
| BCS | same as Branch if LOwer |
| BCHG | test a bit and change it |
| BCLR | clear a bit |
| BSET | set a bit |
| BTST | test a bit |
| CLR | clear a register |
| CMP.B/W/L | compare value and register |
| CMPA.W/L | compare to address register |
| CMPI | compare immediate value |
| CMPM | (address register)+,(address register)+ |
| DIVS | signed 16 bit division |
| DIVU | unsigned 16 bit division |
| EXG | swap any registers |
| EXT | extend the sign |
| EXT.W | extend to bits 8 through 15 |
| EXT.L | extend to bits 16 through 31 |
| JMP | jump to routine |
| JSR | jump to routine; return when completed |
| LEA | Load Effective Address |
| MOVE.B/W/L | move value or register |
| MOVEQ | value is -128 to +127 |
| MOVEI | immediate value |
| MOVEA | move to address register |
| MULS | signed 16 bit multiplication |
| NOP | No OPeration |
| SWAP.W | exchange top and bottom 16 bits in data register |
| TST.B/W/L | compare location or register to 0 |

### SHIFTS - ONLY DATA REGISTERS

| | |
|---|---|
| shift.B/W/L | #1 through #7,data register |
| shift.B/W/L | data register(#1 through #63),data register |
| ASL | CC <— 0 |
| ASR | same —> CC |
| LSL | CC <— 0 |
| LSR | 0 —> CC |
| ROL | CC <===> high bit |
| ROR | bit 0 <===> CC |

| OPCODES | SOURCE,DESTINATION |
|---------|--------------------|
| **BRANCHES** | |
| | GENERAL: |
| BCC | Branch if Carry Clear |
| BCS | Branch if Carry Set |
| BRA | BRanch Always |
| | |
| | UNSIGNED: |
| BEQ | Branch if EQual |
| BHI | Branch if HIgher |
| BHS | Branch if Higher or the Same |
| BLO | Branch if LOwer |
| BLS | Branch if Lower or the Same |
| | |
| | SIGNED: |
| BEQ | Branch if EQual |
| BNE | Branch if Not Equal |
| BGE | Branch if Greater than or Equal |
| BGT | Branch if Greater Than |
| BLE | Branch if Less than or Equal |
| BLT | Branch if Less Than |
| BMI | Branch if MInus |
| BPL | Branch if PLus |

| OPCODES | SOURCE,DESTINATION |
|---------|--------------------|
| **BOOLIAN OPERATIONS** | |
| AND.B/W/L | can't use address registers |
| | must be at least 1 data register |
| ANDI | immediate value number |
| OR.B/W/L | can't use address register |
| | must be at least 1 data register |
| ORI | immediate value number |
| EOR.B/W/L | source must be a data register |
| EORI | immediate value number |
| NEG.B/W/L | reverse the sign |
| NOT.B/W/L | reverse 0's and 1's |

| OPCODES | SOURCE,DESTINATION |
|---------|--------------------|
| **STORAGE** | |
| DS.B/W/L | define storage (amount) |
| DC.B/W/L | define constant |
| Pointer dc.l 0 | |
| Title dc.b 'my title',0 | |
| Array dc.l 0,0,0,0,0 | |
| DCB.B/W/L | define buffer (length, value) |

this command requires the handler location from "Output" to be in register d1, the message or buffer location in d2 and the length of the text to be printed in d3. Any text must be stored in a byte storage area (DC.B) and surrounded by single quotes; text is usually followed by a comma and 0 to indicate the end of text. While "Write" does not check for a 0, the 0 can be used to determine text length. Often a ",10" is included after the text to force a linefeed or jump to the next line, but be sure to count it as a character string and include it in d3.

Let's look at Listing 1 in more detail. As short as it is, this will be the format of most of the programs we'll write—define equates and offsets, set-up, the main program, close-out, and data/locations. It's really pretty easy to understand the general procedure.

The first two offsets I used "OpenLibrary" and "CloseLibrary" are both in the EXEC library; after their offset values, I have included the information that must go in specific registers for the routine to work. The next two offsets are DOS routines and only "Write" requires information in specific registers. I then MOVEd the current starting location to a temporary storage area called "stack," where it will remain until the end of the program. The address of the library name I want to open is stored in register a1 and the latest acceptable version—in this case 0—is stored in d0. Since the routine I want is in the EXEC library and that location is always in address 4, I stored a 4 in register a6. The JSR (Jump to SubRoutine) will transfer the program to a location -552 bytes from the EXEC address and open the DOS library.

The result of the routine is to return the address of the DOS library in register d0. Most routines return the information requested in register d0. We'll store this address in a location called "dosbase." But what if the program couldn't find the library? In that case, register d0 would contain a 0. Check for this with a BEQ (Branch if EQual to 0). The result of the last operation or procedure always gets stored in the CC (Condition Code) register. So if a 0 was returned instead of a library address, the BEQ will branch to "done:".

Put the DOS library address in a6 and then call the "Output" routine saving the CLI console-handler location in "conhandler" and in d1. The location of the message is stored in d2 and its length of 23 characters in d3. Then the DOS "Write" routine is called to print the message; since CHR$(10) is part of the message, there will also be a linefeed and anything else to be printed will go on the next line.

Next, the DOS library location is stored in d0 as a 32 bit binary number. We want to print it, however, as a HEX number with eight characters, each character being 0 through 9 or A through F; this corresponds to CHR$(#$30 through #$39 or #$41 through #$46). Since each HEX number is four bits long (#$F16 = 1111 $_2$), we need to convert each four-bit group into one CHR$ value and put it in our buffer, and we'll need to do this eight times.

The ROL (ROtate Left) command will help us out here. Whenever a byte, word, or long word is rotated, everything shifts over one space in the given direction, but the bit that gets bumped off goes around to the other end. Eight rotations would return a byte to its original value. First we'll store our buffer address in a0 and clear d2 and d3. Now rotate register d0 #4 times to the left; the four left-most bits (31 - 28) are now the four right-most bits (3 - 0). We want to keep using this value so copy it to register d2. Since

all we want are those four bits, AND the register with #$F and everything else will be zeroes. D2 now contains a value from 0 to 15. Convert this to its CHR$ value by adding #$30; if this value is between #$30 and #$39 branch to "ok." If the value is greater though, you'll have to add #7 to get the CHR$ value for A through F. Now, store the string value at the location in register a0 and increase a0 by one byte. The "+" after (a0) will increase that register by the size of the MOVE command. Since we've stored a string value in the buffer, increase the length in d3 by 1; keep doing this until d3 reaches 8. Use the CMP (CoMPare) command to see if a value and a register are the same.

When the length reaches 8 we're almost ready to print the string buffer. As before, put the "conhandler" location in d1, the buffer address in d2, and the length plus #1 (we'll also print a linefeed) in d3. Again, call the "Write" routine and whatever is in the buffer gets printed as an eight-digit HEX number.

After the entire routine, you have to close the library opened at the beginning. "CloseLibrary" is an EXEC routine and requires the library address to be in register a1. Any library that has been opened must be closed before ending the program. Again the EXEC location is stored in register a6 followed by a JSR to the "CloseLibrary" routine. This is followed by "done:". If the "OpenLibrary" routine hadn't worked, the program would have branched to here since there was no opened library. The contents of "stack" are restored to the SP register and the RTS returns to CLI.

The EVEN command will align us on even addresses so that all of the following addresses will be acceptable. Three long word locations are reserved ("stack," "dosbase," and "conhandler") with DC.L 0 and the eight byte buffer is reserved with DCB.B 8,0 (Define Constant Block), followed by a CHR$(10) for a linefeed. After some more EVENs, space for the library name and message are both reserved with DC.B. The message is followed by a CHR$(10) and both are terminated with a 0.

After typing this program, save it as PRINT_D0.ASM. Next, assemble it with A68K PRINT_D0.ASM and when its error-free, blink it with BLINK PRINT_D0.O. After saving the source code and program to the PROGRAMS: disk, type PRINT_D0 and there's your message followed by the DOS library location. You could use this program to print the contents of any register, not just d0. In a future article we'll use some printer codes to add italics, underline, etc.

A68K is very versatile. All of my programs are written with lower-case printing and that is acceptable. If you forget the S (Short), I (Immediate), or A (Address register), A68K will usually add it to the command for you. But try to get into the habit of writing the best program you can and use A68K only to correct mistakes and assemble.

## IN THE FUTURE

In the next article I'll show you how to write some macros that will make assembly even easier. We'll also teach our Amiga how to multiply and divide using fractions and even get it to talk to us! Future articles will discuss graphics as we explore the Mandelbrot/Julia Sets and work our way through arrays. Then it's on to Menus and Gadgets. I would keep re-reading the glossary of commands at the end of this article and add your own notes and comments to it.

Finally, there is no easy way to debug your program. If the problem isn't obvious, set it aside for a while then come back and try again. You could use PRINT_D0 to check the contents of various registers at different times in the program. Usually your error will turn out to be some simple, dumb mistake and you won't believe that you didn't think of it earlier. I try to write the entire program first in Basic and then convert portions of it to assembly. Always save your program and changes before trying them. If you have any questions about these articles you may write to me through the magazine.

## Listing One

```
;LISTING 1
equates:
offsets:
;exec
openlibrary   = -552          ;a1=library name, d0=version
closelibrary  = -414          ;a1=library base
;dos
output        = -60           ;none
write         = -48           ;d1=conhandler, d2=#msg
                              ;d3=length
start:
    move.l   sp,stack          ;save the Stack Pointer
    movea.l  4,a6              ;exec address
    lea      dos,a1            ;desired library
    moveq    #0,d0             ;any version
    jsr      openlibrary(a6)   ;open the library
    move.l   d0,dosbase        ;save the address
    beq      done              ;couldn't find it
    movea.l  d0,a6             ;move address to a6
    jsr      output(a6)        ;output
    move.l   d0,conhandler     ;save location
    move.l   conhandler,d1     ; here, too
    move.l   #msg1,d2          ;message location
    moveq.l  #23,d3            ;message length
    jsr      write(a6)         ;write

;print_lib_location
    move.l   dosbase,d0        ;library address
    lea      buffer,a0         ;text buffer
    moveq    #0,d2             ;clear d2
    moveq    #0,d3             ; and d3
more:
    rol.l    #4,d0             ;move left 4 bits to right
    move.l   d0,d2             ;resave in d2
    and.b    #$f,d2            ;get just 4 bits
    addi.b   #$30,d2           ;character $ number
    cmpi.b   #$3a,d2           ;greater than 9?
    bcs.s    ok                ;branch if not
    addq.b   #7,d2             ;add 7 to get A-F
ok
    move.b   d2,(a0)+          ;store in buffer
                              ;increase buffer 1 byte
    addq.l   #1,d3             ;increase d3
    cmpi.l   #8,d3             ;reached 8 yet?
    bne.s    more              ;branch if not
    move.l   conhandler,d1     ;output file location
    move.l   #buffer,d2        ;buffer location
    addq.w   #1,d3             ;adjust for linefeed
    jsr      write(a6)         ;print buffer+linefeed

    movea.l  a6,a1             ;dos library location
    movea.l  4,a6              ;exec library location
    jsr      closelibrary(a6)  ;close dos library
done:
    move.l   stack,sp          ;return Stack Pointer
    rts                        ;return to CLI
    even
stack       dc.l  0            ;stack storage
dosbase     dc.l  0            ;library location
conhandler  dc.l  0            ;output file location
buffer      dcb.b 8,0          ;hex buffer
    dc.b  10                   ;linefeed
    even
dos         dc.b  'dos.library',0  ;library name
    even
msg1        dc.b  'This is a test message',10,0
    end
```

# Programming
# the Amiga's GUI* in C—Part IV

* Graphical User Interface

by  Paul Castonguay

## Intuition's Border Function

Due to popular request, this series, which was originally intended to last only four issues, is being extended. I thank those readers who have mailed in letters of support. I also encourage others who would like to read about certain aspects of the Amiga's high-level operating system to make their requests known.

This issue contains a pleasant new twist. Somewhere in this text you will find a small programming challenge, a puzzle if you will. The first five readers to best solve the puzzle as described will win a one year subscription (or renewal) of AC's TECH Magazine. Mail entries to:

> Paul Castonguay
> P.O. Box 505
> Everett, MA 02149

I have added this new feature for the purpose of promoting more "Programming as a Hobby" on the Amiga. That's right, some people bought their Amigas to have fun programming it. Why not? Compared to other platforms, it is a programmer's paradise.

In this issue you will find:

1. A Discussion of Intuition's high-level line drawing function, DrawBorder( ).
2. Two implementations of Rosettes using DrawBorder( ).
3. A discussion of Intuition's low level text rendering function, Text( ).
4. A discussion of Intuition's high level text rendering function, PrintIText( ).

The example programs in this issue all use the programming shell that we have developed over the course of the last three articles. For your convenience, a copy of it has been placed in each example directory, along with a make file (lmkfile). If you make modifications to the programs you can conveniently re-compile by entering LMK on the AmigaDOS command line, or by double-clicking the "Build" icon on the WorkBench.

In this article, I use the term "line drawing" to mean a graphic image that is constructed of straight lines drawn between any number of defined points. In a future article I will discuss another kind of graphic image, one that is defined directly by bitplane data.

With the last issue you began drawing some simple images using the Amiga's primitive functions: WritePixel( ), Move( ), and Draw( ). These are probably the most often used graphic functions on the Amiga for creating line drawings. WritePixel( ) illuminates

individual pixels on the screen, Move( ) places the graphic cursor (or drawing point) at any desired location, and Draw( ) connects two points with a straight line. To create complex drawings using these functions, you write a series of instructions that mimic how you would produce the same thing yourself using pencil and paper.

## Speeding Up the Drawing Process

Often the points of a drawing are based on some equation and require floating point arithmetic for their calculation, a task that is more time consuming than integer arithmetic. In some cases we can overcome this slowness by using a programming trick: storing the calculated, pixel coordinate values of all the points in a drawing in an integer array. I like to call this array the points-array. Once that is done, the Amiga's graphic functions can render the drawing at any time by using these values directly, without having to perform any further floating point calculations.

A good example where this concept can be used to an advantage is in the drawing of a rosette, which is built up by drawing a large number of straight lines between some smaller number of pre-defined points, arranged at equal distances around the circumference of a circle. A 23-point rosette consists of some 253 lines. Drawing all those lines using the original equations from which the points were defined, along with some reasonably adequate scaling functions, would require over 2000 floating point calculations. Instead we can calculate the pixel numbers of those 23 points once and store them in an integer array. Then the repetitive invocations of the Draw( ) function, which produces the actual lines of the rosette, can use those pre-calculated values directly.

A version of the rosette program is reproduced for you this month on the magazine disk, in the directory Rosette1. Observant readers will notice that I have added a macro definition, POINTS, to allow you to conveniently change the number of points in the drawing. I have also increased the size of the point-array (Rosette_Points), to allow you to more conveniently draw rosettes having different point sizes. Now, compare the operating speed of that program with the one in the directory called Rosette0. That difference in speed is due to the fact that the one in Rosette0 is performing floating point calculations for every line that it draws, while the other is using the above programming concept.

## Intuition's High Level Drawing Function

Intuition's high level, line drawing function, called DrawBorder( ), can produce line drawings using this same concept of storing pre-calculated pixels values in an array. It renders by drawing straight lines between points stored in the array sequentially, that is, in tandem.

As with most high-level operations in Intuition, there is a general operational theme to follow when using DrawBorder( ):

1. Declare one or more Border structures to contain the specifications of your drawing, things like its color, how many points it has, ... etc. I like to call this the description structure.
2. Store the pre-calculated pixel numbers representing all the points in your drawing in an array such that drawing straight lines between them sequentially will reproduce it. This may require that some points be stored a multiple number of times at different positions in the array. I call this array the Border-data-array, to discriminate it from the one we used earlier in our rosette example. The Border-data-array must be located in chip RAM, a special section of memory that is addressable by the Amiga's graphic co-processor chips.
3. Link all structures and their respective Border-data- arrays together.
4. Invoke the DrawBorder( ) function.

This general plan is a bit more complicated than using the primitive Move( ) and Draw( ) functions directly. However, it has certain structural advantages, depending on exactly what kind of drawing you want to produce, and it can render complex drawings very fast. I might also add at this point that the above operational theme is similar to that used in another Intuition, high-level function that I will discuss today, PrintIText( ).

## The Border (Description) Structure

The template definition for the Border structure is in the file <intuition/intuition.h>. I reproduce it below without its comments:

```
struct Border
{
        SHORT   LeftEdge;
        SHORT   TopEdge;
        UBYTE   FrontPen;
        UBYTE   BackPen;
        UBYTE   DrawMode;
        BYTE    Count;
        SHORT   *XY;
        struct  Border *NextBorder;
};
```

LeftEdge and TopEdge refer to the position of the drawing, measured in pixels from those edges of its graphic window. I'll have more to say about this later. FrontPen is the color register (pen number) that you wish to use. There is no need to call SetAPen( ), as with the primitive graphics functions. Also, DrawBorder( ) does not affect the current setting of SetAPen( ).

BackPen is currently unused by the DrawBorder( ) function.

There are two ways that DrawBorder( ) can render drawings. The first is called JAM1. This term means two things. First, that the drawing will be reproduced using one color per Border structure, as specified in the FrontPen member, and second, that the drawing will overwrite any previously drawn graphic images.

This method effectively blasts your image down on the screen, obliterating everything in its path. Note that JAM1 does not restrict you to only one color per drawing. Multiple colored drawings are rendered by building up multiple Border, graphic structures, as you will soon see.

The second way that DrawBorder( ) renders drawings is called COMPLEMENT. In this mode the color is determined not by FrontPen, but by the binary complement of the drawing surface. For example, if you are drawing on an eight color, blank screen, meaning that all pixels are color 0, then your image will be drawn in the bitwise complement of 0, which is color 7. However—and this is the interesting part—if your drawing overlaps any portions of previously drawn images or text, then those related pixels will be replaced, not by color 7, but by whatever color is their bitwise complement. The following chart lists bitwise complement colors for an eight-color screen.

| Color Register | Binary | Bitwise Complement | COMPLEMENT Color |
|---|---|---|---|
| 0 | 000 | 111 | 7 |
| 1 | 001 | 110 | 6 |
| 2 | 010 | 101 | 5 |
| 3 | 011 | 100 | 4 |
| 4 | 100 | 011 | 3 |
| 5 | 101 | 010 | 2 |
| 6 | 110 | 001 | 1 |
| 7 | 111 | 000 | 0 |

The COMPLEMENT drawing mode has the interesting property that it allows you to erase your image by drawing it again! But this is an advanced concept. For now, let's stick to the easier-to-use JAM1 DrawMode.

The count member of the Border description structure refers to the number of points (coordinate pairs) stored in the Border-data-array. It represents the number of points between which straight

lines will be sequentially drawn. This member is type BYTE and therefore cannot store values greater than 255. Thus 255 is the maximum number of points that the DrawBorder( ) function can render from a single Border structure. Again, this is not a limitation since larger drawings can be produced by using multiple Border, graphic structures. The XY member is a pointer to the drawing's Border-data-array. In a minute, you will see two methods for declaring that array so that it properly resides in chip RAM.

The last member of the border description structure is a pointer to another Border structure. You use this member when you want to render complex drawings consisting of more that one Border structure, drawings that have more than one color, that consist of several discontiguous sets of lines, or that have more than 254 lines. In our first example, we will use only one structure, in which case this member is assigned a NULL.

## The Border-Data Array

The purpose of this array is to store the pre-calculated pixel numbers of the various points of your drawing in such an order that drawing straight lines between them sequentially will reproduce the drawing. The array itself must be accessible to the Amiga's graphic co-processor chips, and therefore must be declared in chip RAM. SAS/C makes that easy to do with their platform specific data type called chip.

```
SHORT chip MyData[] = {   0, 0,
                        100, 0,
                        100, 50,
                          0, 50,
                          0, 0
                       };
```

The above array is declared in chip RAM and is initialized to the coordinates of a rectangle that is 100 pixels wide, and 50 pixels high. The data is in pixel numbers measured from the reference point defined in the LeftEdge and TopEdge members of the above Border structure. The SAS/C chip data type must be declared in the global section of your program, outside main( ). You don't have to worry about returning this memory to the operating system before your program terminates; the compiler takes care of that for you automatically.

## A Dynamic Solution

The above method is called static in the sense that it is fixed for only one size of drawing, in this case one having four lines. Suppose you don't know, in advance, exactly how many lines are in your drawing. Quick, tell how many are in a 27-point rosette? Do you see what I mean? You could of course write code to calculate and report that answer easily enough, but in C you are not allowed to declare an array of variable size using the above notation. So even though your program is able to calculate the number of lines it needs to draw, it cannot declare a Border-data-array of the proper size to do so.

You could solve the above dilemma by declaring your Border-data-array large enough to handle any drawing that comes along, but that would be impractical. Complex drawing are rendered by multiple Border, graphic structures, and their exact characteristics are impossible to predict in advance.

The solution is to allocate memory dynamically, using exec's AllocMem( ) function. It requires two arguments, the number of bytes that you need, which can be calculated by your program, and a macro specifying the kind of memory needed, in this case MEMF_CHIP. You can also use the convenient macro MEMF_CLEAR to initialize the array to zero.

```
BorderData = (SHORT *)AllocMem(bytes, MEMF_CHIP | MEMF_CLEAR);
```

Whether your programs are complex graphics, or data base applications, allocating memory dynamically is the most efficient way to design them. They will use only as much memory as they need, when they need it. In addition, you will be able to allocate chip memory from within different functions, not just globally as in SAS/C's chip data type. However, this method does have the disadvantage that it is slightly more difficult to use. Your programs can often develop some difficult-to-dentify bugs. Simple errors can produce system crashes.Save your work often!

The following code declares some memory and assigns to it the same data as the above rectangle example. Note that I used 5*2 to specify the amount of memory, emphasizing that I need enough for 5 coordinate pairs.

```
SHORT *MyData = NULL;
SHORT *ptr = NULL;

MyData = (SHORT *)AllocMem(5*2*sizeof(SHORT), MEMF_CHIP | MEMF_CLEAR);

ptr = MyData;

*ptr++ =   0; /* point 1 */
*ptr++ =   0;

*ptr++ = 100; /* point 2 */
*ptr++ =   0;

*ptr++ = 100; /* point 3 */
*ptr++ =  50;

*ptr++ =   0; /* point 4 */
*ptr++ =  50;

*ptr++ =   0; /* point 5 */
*ptr   =   0;
```

This method of declaring the Border-data-array also gives you the responsibility of returning the memory to the operating system before your program terminates, using exec's FreeMem( ) function.

## Linking Data to the Description Structure

For Intuition to properly execute the DrawBorder( ) function, the Border-data-array must be linked to the Border description structure representing the drawing. This is easily done with an assignment instruction. Below I give you a hierarchical diagram of what I have done so far:

```
                                          MyData array
                                          (in chip ram)

   Border Structure
                                         ┌──────────────┐
  ┌─────────────────────────┐            │  0,     0,   │
  │ LeftEdge    = 0          │            │              │
  │ TopEdge     = 0          │            │ 100,    0,   │
  │ FrontPen    = 1          │            │              │
  │ DrawMode    = JAM1       │            │ 100,   50,   │
  │ Count       = 5          │            │              │
  │ XY ──────────────────────────────────▶│  0,    50,   │
  │ NextBorder  = NULL       │            │              │
  └─────────────────────────┘            │  0,     0,   │
                                         └──────────────┘
```
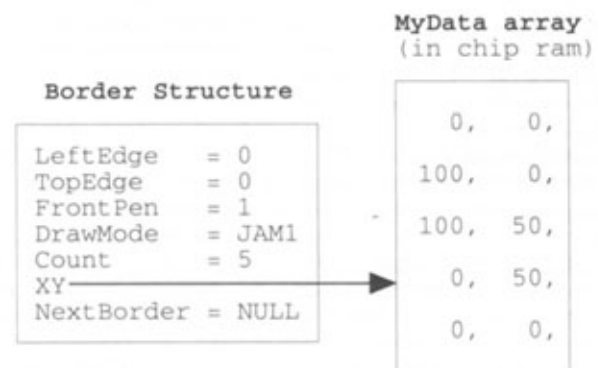
## The Drawborder( ) Function

After performing all the above ground work, the only thing left to do is invoke the DrawBorder( ) function itself:

```
DrawBorder(my_rp, &MyBorder, x, y);
```

This function renders the rectangle by drawing straight lines between the points stored in the array MyData. It does this sequentially, as you would yourself if you were drawing while not being allowed to lift your pencil off the paper. To produce a closed image the first and last points must coincide.

The first argument in the DrawBorder( ) function is the forever ubiquitous RastPort pointer. The system needs it to determine where you want your drawing rendered, in this case the window opened by your programming shell. The second argument is the address of the Border structure that describes the drawing you want rendered. The system will read from it your specifications, as well as the actual data points that you have previously linked to it via its XY pointer. The last two arguments are an offset coordinate pair that will get added to the position of every point in your drawing.

## Position Control

You have three levels of control for the positions of points in your drawing: the LeftEdge and TopEdge members of the Border structure, the individual coordinates in the Border-data-array, and finally the offset coordinates of the DrawBorder( ) function.

LeftEdge and TopEdge represent a coordinate pair that act as a reference for all points controlled by that structure. This level of control is used when constructing multiple Border, graphic structures, to adjust the relative positions of different parts of the drawing. The second level of control is the pixel data itself, which represents positions relative to the above LeftEdge and TopEdge. Finally the DrawBorder( ) offset value allows you to make one final adjustment to the position of the entire drawing. This is especially useful for adjusting the position of multiple Border, graphic images, as you will soon see.

## First Border Example

This first example draws the above-mentioned rectangle. It is a single Border structure drawing; thus its NextBorder member is assigned a NULL. The example is on the magazine disk in the directory called First_Border. I list it below:

```
/* First_Border.c

    Paul Castonguay                          October 1991
    ================================================================ */
#include <intuition/intuition.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/dos.h>

#include "Shell.h"

SHORT chip MyData[] = {   0, 0,
                        100, 0,
                        100, 50,
                          0, 50,
                          0, 0
                      };
struct Border MyBorder = {
                            0,      /* SHORT LeftEdge      */
                            0,      /* SHORT TopEdge       */
                            1,      /* UBYTE FrontPen      */
                            0,      /* UBYTE BackPen       */
```

```
                            JAM1,   /* UBYTE DrawMode          */
                            5,      /* BYTE Count              */
                            MyData, /* SHORT *XY               */
                            NULL    /* struct Border *NextBorder */
                        };

VOID main(int argc, char *argv[]);

VOID main(int argc, char *argv[])
{
    struct Screen *my_screen;
    struct ViewPort *my_svp;
    struct Window *my_window;
    struct RastPort *my_rp;

    SHORT i;

    if(!Open_Shell(&my_screen, &my_svp, &my_window, &my_rp, "LACEHIGH8"))
    {
        printf("Problems in Open_Shell( )\n");
        Delay(100);
        exit(RETURN_FAIL);
    }

    /* ================= Your code starts here ================= */

    for(i = 0; i < 5; i++)
        DrawBorder(my_rp, &MyBorder, i*80, i*40);

    SetAPen(my_rp, 1);
    Move(my_rp, 200, 10);
    Text(my_rp, "Rectangles using DrawBorder( )", 29);

    Delay(500);

    /* =============== Close shell before quitting =============== */

    Close_Shell(my_window, my_screen);
}
```

This is a static solution. The Border-data-array is declared globally using SAS/C's chip data type. Notice that the drawing's Border structure is also declared globally, although strictly speaking that was not necessary. I did so in order to keep the two related declarations close together in the listing. The program contains a loop that draws several instances of the rectangle using DrawBorder( )'s offset feature. What you have here is five drawings produced from a single Border, graphic structure by multiple invocations of the DrawBorder( ) function, each at a different position on the screen.

The next example, Second_Border.c, demonstrates the same program except that the Border-data-array is declared dynamically and private to main( ). Note that two new #include's were needed: <exec/memory.h> which contains the macro definition for MEMF_CHIP, and <proto/exec.h> which contains the ANSI prototype definitions of the AllocMem( ) and FreeMem( ) functions.

```
/* Second_Border.c

    Paul Castonguay                          October, 1991
    ================================================================ */
#include <intuition/intuition.h>
#include <exec/memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/dos.h>
#include <proto/exec.h>

#include "Shell.h"

VOID main(int argc, char *argv[]);

VOID main(int argc, char *argv[])
{
    struct Screen *my_screen;
    struct ViewPort *my_svp;
    struct Window *my_window;
```

```
                  MyBorder[0]

       LeftEdge     = 0
       TopEdge      = 0
       FrontPen     = 1
       DrawMode     = JAM1
       Count        = 5
       XY ───────────────┐
       NextBorder   = NULL │
                           │              MyData array
                           │             (in chip ram)
                  MyBorder[1]
                                         ┌─────────────┐
       LeftEdge     = 80                 │  0,    0,   │
       TopEdge      = 40                 │             │
       FrontPen     = 2                  │ 100,   0,   │
       DrawMode     = JAM1   ──────────▶ │             │
       Count        = 5                  │ 100,  50,   │
       XY ─────────────────────────────▶ │             │
       NextBorder   = NULL   ──────────▶ │  0,   50,   │
                                         │             │
                                         │  0,    0,   │
                  MyBorder[2]            └─────────────┘

       LeftEdge     = 160
       TopEdge      = 80
       FrontPen     = 3
       DrawMode     = JAM1
       Count        = 5
       XY ───────────────────┘
       NextBorder   = NULL

       ··· (last structure in list points to NULL)
```
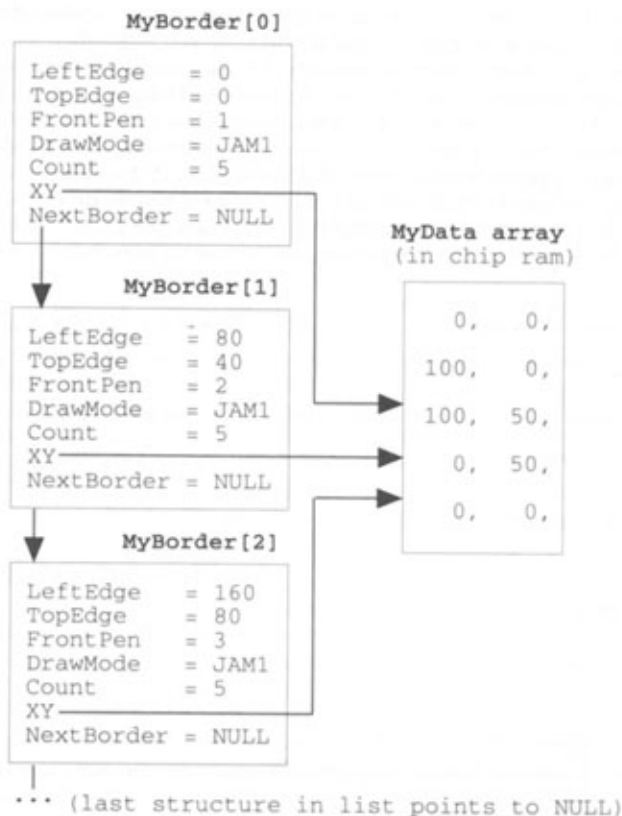
## Rosettes Using DrawBorder()

The DrawBorder( ) function is not limited to simple drawings. Let me demonstrate by using it to render a rosette. Recall that I introduced the rosette early in this series, promising to use it later to demonstrate different programming concepts. It's more interesting than rectangles, don't you think?

In order to make the program capable of efficiently drawing rosettes of different point sizes, I will declare whatever memory it needs dynamically. The larger the rosette, the more memory it will use for its graphic data structures. Also you will see that the amount of memory used depends on how ingeniously we design the program.

Now, exactly how can we design a bunch of linked Border structures to represent a rosette? Perhaps we could divide the rosette into groups of lines having some similar property? Well, that could work in principle. A natural division might be in groups that connect to one common point. Each group would represent a fan pattern of lines. The trouble is, DrawBorder( ) draws lines in tandem, connecting points that are stored in a Border-data-array sequentially. That's great for drawing closed figures, like polygons, but not very efficient for fan patterns. The computer would have to trace back over each line that it drew in order to get back to their common point before drawing the next one. That's too time consuming.

One answer is to divide the rosette up into individual lines, making a separate Border structure for each. That's an awful lot of Border structures, but the Amiga can handle it easily. All these structures must be linked together and each one linked to four data values (two points, each having horizontal and vertical coordinates) stored in chip RAM. Sound complicated? It's easier than you think,

and a good demonstration of Intuition's high level power as well. The solution is on disk in the directory Rosette2. I explain its internal operation below.

The first order of business is to calculate the number of lines in the rosette. To do that we use the same double nested loop that normally generates it, except that here we only increment a line counter:

```
/* ================================================ */
/* === Calculate number of lines in drawing === */
/* ================================================ */
for(i = 0; i < POINTS; i++)
    for(j = i+1; j < POINTS; j++)
        line_count++;
```

You can put a print instruction after this loop and confirm that the exact number of lines in a 23-point rosette is 253. Next we allocate memory for our graphic structures:

```
/* =================================================== */
/* === Allocate Border-data-array in chip ram === */
/* =================================================== */
Rosette_Data = (SHORT*)AllocMem(4*line_count*sizeof(SHORT),
                MEMF_CHIP | MEMF_CLEAR);

/* =================================================== */
/* === Allocate Border structure array === */
/* =================================================== */
Rosette_Border = (struct Border *)
                AllocMem(line_count*sizeof(struct Border),
                MEMF_CLEAR);
```

The first declaration, Rosette_Data, will be used to store our pixels numbers. It is one contiguous length of chip memory, long enough to store 4 four integers per line of the rosette. The second memory declaration is for the array of Border structures, one for each line. This one need not be in chip RAM. Each Border structure will have to point to a different spot in the previously declared length of chip RAM Rosette_Data.

Now we need to calculate the pixel numbers for the points of the rosette. We use a point-array for this, for the same reason that we did in our earlier rosette examples, to keep the number of floating point calculations in our program as low as possible. Note that this is not the data that DrawBorder( ) will use directly, but rather the data that we will use to fill up chip memory with the data that DrawBorder( ) will use.

```
/* =================================================== */
/* === Calculate and store points around circle === */
/* =================================================== */
for(i = 0; i < POINTS; ++i)
{
    Rosette_Points[i][0] = Fx(cos(Rad(Angle)));
    Rosette_Points[i][1] = Fy(sin(Rad(Angle)));
    Angle += 360.0/POINTS;
}
```

We then assign data to chip memory using these pre-calculated values, thus:

```
/* =================================================== */
/* === Loop through chip ram assigning end points === */
/* =================================================== */
Data_ptr = Rosette_Data;
for(i = 0; i < POINTS; ++i)
    for(j = i; j < POINTS; ++j)
    {
        *Data_ptr++ = Rosette_Points[i][0];
        *Data_ptr++ = Rosette_Points[i][1];
        *Data_ptr++ = Rosette_Points[j][0];
        *Data_ptr++ = Rosette_Points[j][1];
    }
```

```
struct RastPort *my_rp;

SHORT *MyData = NULL;
SHORT *ptr = NULL;
SHORT i;
struct Border MyBorder;

if(!Open_Shell(&my_screen, &my_svp, &my_window, &my_rp, "LACEHIGH8"))
{
    printf("Problems in Open_Shell( )\n");
    Delay(100);
    exit(RETURN_FAIL);
}

/* ================== Your code starts here ================== */

MyData = (SHORT *)AllocMem(5*2*sizeof(SHORT), MEMF_CHIP | MEMF_CLEAR);

ptr = MyData;

*ptr++ = 0;
*ptr++ = 0;

*ptr++ = 100;
*ptr++ = 0;

*ptr++ = 100;
*ptr++ = 50;

*ptr++ = 0;
*ptr++ = 50;

*ptr++ = 0;
*ptr   = 0;

MyBorder.LeftEdge   = 0;
MyBorder.TopEdge    = 0;
MyBorder.FrontPen   = 1;
MyBorder.BackPen    = 0;
MyBorder.DrawMode   = JAM1;
MyBorder.Count      = 5;
MyBorder.XY         = MyData;
MyBorder.NextBorder = NULL;

for(i=0; i<5; i++)
    DrawBorder(my_rp, &MyBorder, 80*i, 40*i);

Delay(500);

FreeMem(MyData, 5*2*sizeof(SHORT));

/* ================== Close shell before quitting ================== */

Close_Shell(my_window, my_screen);
}
```

## Drawings in Multiple Colors

Exactly what method you use to declare your array will depend on the structure of your program. If you are drawing something whose definition is known at the time you are writing the program, and which is used in several places within your program, it may be more convenient for you to use the global version. However, if you must rely on internal calculations to define points in your drawing, you will need to use the dynamic method. But even if your drawing is completely known ahead of time, you may still need to use the dynamic method. Any program that is reasonably complex should be structured into functions and have its graphic structures made local, in order that you can come to grips with its overall complexity. Program structure is not a strict rule that you must follow to get your programs working, but a feature that you should use to make them maintainable. Authors of unstructured programs lead lonely lives because no one can understand their work.

### Drawings in Multiple Colors

If you have a drawing that consists of a single border structure and you want to render it in different colors, and at different locations on the screen, you have a number of choices. You could

change the FrontPen member of the Border description structure before every invocation of DrawBorder( ), which might then render it using different offset coordinates. Or, you could create several Border structures, each having the same LeftEdge and TopEdge coordinates, pointing to the same Border-data-array, but having different FrontPen definitions. Then each invocation of DrawBorder( ) would specify the address of a different Border structure, as well as different offset coordinates of course. An easy way to do that is to declare an array of Border structures, as in example Third_Border.c on the magazine disk. I reproduce the important parts below

```
SHORT *MyData = NULL;
SHORT *ptr = NULL;
SHORT i;
struct Border MyBorder[5];

MyData = (SHORT *)AllocMem(5*2*sizeof(SHORT), MEMF_CHIP | MEMF_CLEAR);

ptr = MyData;

*ptr++ =   0;
*ptr++ =   0;
*ptr++ = 100;
*ptr++ =   0;
*ptr++ = 100;
*ptr++ =  50;
*ptr++ =   0;
*ptr++ =  50;
*ptr++ =   0;
*ptr   =   0;

for(i=0; i<5; i++)
{
    MyBorder[i].LeftEdge   = 0;
    MyBorder[i].TopEdge    = 0;
    MyBorder[i].FrontPen   = i+1; /* different color for each */
    MyBorder[i].BackPen    = 0;
    MyBorder[i].DrawMode   = JAM1;
    MyBorder[i].Count      = 5;
    MyBorder[i].XY         = MyData;
    MyBorder[i].NextBorder = NULL;
}

for(i=0; i<5; i++)
    DrawBorder(my_rp, &MyBorder[i], i*80, i*40);
```

Note that in each case the DrawBorder( ) function renders a drawing represented by a different Border structure, but having the same pixel data. The Border structures are not linked.

### Linked Border Structures

Another way to display colored rectangles is to declare several Border structures, each one having its own FrontPen color, as well as its own LeftEdge and TopEdge coordinates, and then link them all together using the NextBorder member. This is your first example of a drawing represented by a multiple Border, graphic structure. The structures are linked sequentially by having the NextBorder member of each point to the Border structure of another, with the last one pointing to NULL. This is the old linked list idea that you learn about in any computer-science data-structures course.

The big difference in this example is that only a single invocation of DrawBorder( ) is required to render all five rectangles. See example Fourth_Border.c on the magazine disk, which draws two complete drawings, 10 rectangles, using two invocations of DrawBorder( ), each using different offset coordinates. Below I give a partial hierarchical diagram of the structure which represents the drawing *(next page)*:

The Data_ptr pointer jumps along the contiguous length of chip memory assigning pixel coordinates that represent end points of all the lines in the rosette. The last piece of work is the assignment and linkage of the Border structures:

```
/* ========================================= */
/* === Assign and link border structures === */
/* ===          and Border-data-arrays   === */
/* ========================================= */
Rosette_Border_ptr = Rosette_Border;

for(i = 0; i < line_count; i++)
{
    Rosette_Border_ptr->LeftEdge    = 0;
    Rosette_Border_ptr->TopEdge     = 0;
    Rosette_Border_ptr->FrontPen    = 5;
    Rosette_Border_ptr->BackPen     = 0;
    Rosette_Border_ptr->DrawMode    = JAM1;
    Rosette_Border_ptr->Count       = 2;
    Rosette_Border_ptr->XY          = (Rosette_Data + i*4);

    if(i < line_count-1)
    {
      Rosette_Border_ptr->NextBorder = Rosette_Border_ptr+1;
      Rosette_Border_ptr++;
    }
    else
        Rosette_Border_ptr->NextBorder = NULL;
}
```

We use a pointer, Rosette_Border_ptr, to jump along the section of memory representing the structures, assigning members as we go along. Each XY member is linked to the pixel data of its respective line in chip memory, four integers representing two coordinates on the screen. The Count member of each Border structure is only 2, meaning that each one will draw a single line between two points. The last Border structure is made to point to NULL, representing the end of the linked list.



**Rosette_Border[0]**

```
LeftEdge    = 0
TopEdge     = 0
FrontPen    = 5
DrawMode    = JAM1
Count       = 2
XY————————————————▶
NextBorder  = NULL
```

**Rosette_Border[1]**

```
LeftEdge    = 0
TopEdge     = 0
FrontPen    = 5
DrawMode    = JAM1
Count       = 2
XY————————————————▶
NextBorder  = NULL
```

**Rosette_Border[2]**

```
LeftEdge    = 0
TopEdge     = 0
FrontPen    = 5
DrawMode    = JAM1
Count       = 2
XY————————————————▶
NextBorder  = NULL
```

**Rosette_Data**

```
4
Integers
```

```
4
Integers
```

```
4
Integers
```

...

... (last structure in list points to NULL)
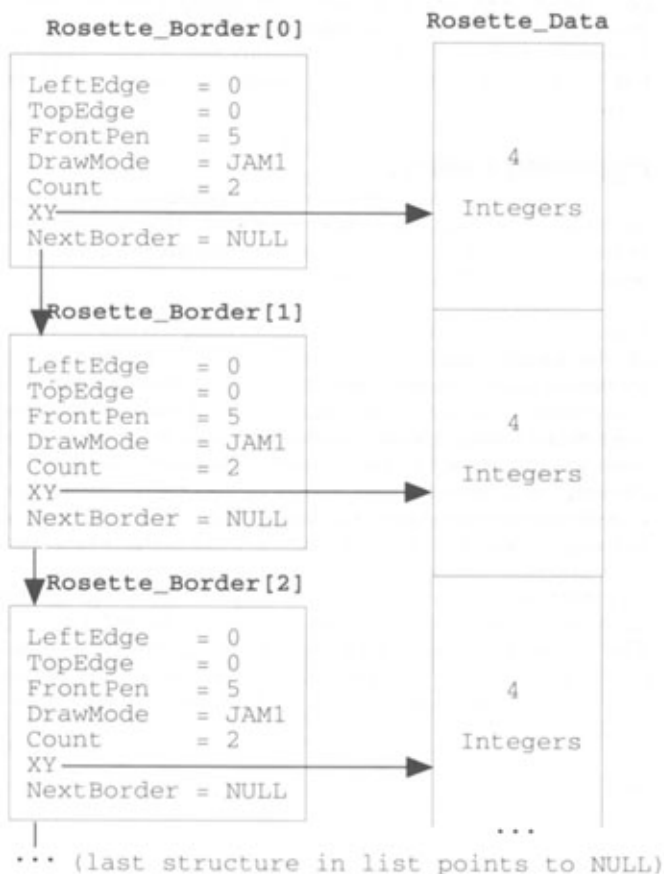
All that done, we can now invoke a single DrawBorder() instruction:

```
DrawBorder(my_rp, &Rosette_Border[0], 0, 0);
```

... and presto, our rosette gets drawn. Like magic.

Perhaps you are thinking that all of this is too complicated to be useful. To draw a single rosette? Perhaps. But what if you wanted to draw many rosettes, perhaps at different places on the screen. Once you have constructed the graphic structures, a drawing can be rendered at any time by a single DrawBorder( ) instruction. Example Rosette3 on disk does that. First it scales the screen such that the origin is in the center of the top left quadrant of the screen.

```
s.xmin = -1.4;
s.xmax = 4.2;
s.ymin = -3.0;
s.ymax = 1.0;

if(!Scale_Window(my_window))
{
    Close_Shell(my_window, my_screen);
    exit(RETURN_WARN);
}
```

Then the graphic structures are constructed as before. Finally four different Rosettes are rendered like so:

```
DrawBorder(my_rp, &Rosette_Border[0], 0, 0);
DrawBorder(my_rp, &Rosette_Border[0], Fx(2.8)-Fx(0.0), 0);
DrawBorder(my_rp, &Rosette_Border[0], 0, Fy(-2.0)-Fy(0.0));
DrawBorder(my_rp, &Rosette_Border[0], Fx(2.8)-Fx(0.0), Fy(-2.0)-Fy(0.0));
```

I wanted to use the Fx( ) and Fy( ) scaling functions to make the placement of each rosette easier to visualize. To do that, I had to remember that the offset argument of DrawBorder( ) is added, not to my Cartesian coordinates, but to the top left corner of the window in which my drawing is being rendered. The graphic data structures were set up such that the center of the rosette is at Cartesian coordinates (0,0), which corresponds to pixel coordinates Fx(0.0), Fy(0.0). To shift the rosette to a different position requires that I subtract those Fx(0.0) and Fy(0.0) terms from each new location.

Rosettes are only an example that I chose to use because they are slightly more interesting than rectangles. The purpose of all this is to help you learn how to use the DrawBorder() function. I really don't know what kind of drawing you will want to render. But whatever it is, you will probably want to build it up in a structured manner, using a different Border structure for each section of the drawing.

### Another Implementation of Rosette

Although the above example is certainly a good one for showing a complex graphic structure, it does not make most efficient use of memory. The problem has to do with the repetition of coordinate values in the Border-data-arrays. You see, each line is defined by two end points and consequently each Border structure points to a four-element array. But often the beginning point of one line is the same as the end point of another. One would think that different Border structures could share the data for those common points. But the above algorithm does not take advantage of that. To design a program that does, we could divide the lines of the rosette into groups of contiguous lines, that is, lines that consist of several segments drawn in tandem. Then points that are common to two segments would not have to be repeated.

Suppose you start at one point on the rosette and draw a line to another immediately adjacent one. Now, moving in the same direction draw another line, this time not to the next immediately adjacent but to the one after that. That is, skip a point. Continuing,

draw another line, this time skipping two points, then another, this time skipping three, ... etc. The result will be a kind of spiral that traces a unique path through the rosette. How many such steps can be taken by a single line? In a 23-point rosette, only 11. After that the line is no longer unique and will start drawing certain lines for the second time. However, if you go back to the point where that line started and move over one point, you can begin another similar, 11-segment unique line. And, how many times can you do this? You guessed it, 23 times! Thus you can complete the entire rosette by drawing 23 contiguous lines, each one consisting of 11 sections.

The example program in the directory called Rosette2A solves the rosette using this new technique. Its graphic structure consists of 23 linked Border structures, each rendering a unique 11-segment path through the rosette. The example demonstrates how ingenious methods can be used to reduce the size of the graphic structures. It also demonstrates the perils of dynamic programming. That's right, the example is programmed dynamically and will render rosettes of any size up to 49 points simply by changing the macro POINTS.

Here is the memory allocation for the Border_data_array:

```
/* =============================================== */
/* === Allocate pixel-data-array in chip ram === */
/* =============================================== */
Rosette_Data=(SHORT *)AllocMem(2*(line_count+POINTS)*sizeof(SHORT),
                MEMF_CHIP | MEMF_CLEAR);
```

We multiply the line_count by 2, not 4, because this method takes advantage of using common points to draw contiguous lines. But watch out for the trap. You must add 23 extra points (POINTS) because there are 23 separate line sets, each having 2 end points which are not shared. Fail to see that and you will crash your machine.

Here is the memory allocation for the Border structures:

```
/* =============================================== */
/* === Allocate Border structure array === */
/* =============================================== */
Rosette_Border = (struct Border *)
                AllocMem(POINTS*sizeof(struct Border),
                MEMF_CLEAR);
```

Next we must assign data to the Border_data_array,

```
Data_ptr = Rosette_Data;    /* Beginning of chip ram array */

/* =============================================== */
/* === Loop through each Border_data_array === */
/* ===       calculating and allocating      === */
/* ===          different line sets          === */
/* =============================================== */
for(line_set = 0; line_set < POINTS; line_set++)
{

  /* =============================================== */
  /* === Assign first point in this line set === */
  /* =============================================== */
  *Data_ptr++ = Rosette_Points[line_set % POINTS][0];
  *Data_ptr++ = Rosette_Points[line_set % POINTS][1];

  points = 1;    /* total point allocations per Border structure */
  step = 1;      /* jump size                                    */
  i = 0;         /* jump variable                                */

  do
  {
    i += step;   /* jump ahead one step */

    /* =============================================== */
    /* === Assign points in this Border structure === */
    /* =============================================== */
    *Data_ptr++ = Rosette_Points[(line_set+i) % POINTS][0];
    *Data_ptr++ = Rosette_Points[(line_set+i) % POINTS][1];

    points++;    /* total point allocations per Border structure */
    step++;      /* increase jump size                           */

  } while(points <= line_count/POINTS);
```

We pick a point on the rosette, line_set, then increment by ever increasing step sizes until 11 line segments are completed (line_count/POINTS). Notice how I use the modulus operator, %, to wrap around through values in the Rosette_Points array. We then increment line_set and start the next contiguous 11-segment line.

That done, we must now link the 23 Border structures to each other, and to their respective Border-data-arrays.

```
/* =============================================== */
/* === Assign and link border structures === */
/* === structures and pixel-data-arrays === */
/* =============================================== */

Rosette_Border_ptr = Rosette_Border; /* Beginning of structures */

for(i = 0; i < POINTS; i++) {
  Rosette_Border_ptr->LeftEdge  = 0;
  Rosette_Border_ptr->TopEdge   = 0;
  Rosette_Border_ptr->FrontPen  = 5;
  Rosette_Border_ptr->BackPen   = 0;
  Rosette_Border_ptr->DrawMode  = JAM1;
  Rosette_Border_ptr->Count     = points;
  Rosette_Border_ptr->XY        = Rosette_Data + i*2*points;

  if(i < POINTS-1)
  {
    Rosette_Border_ptr->NextBorder = Rosette_Border_ptr+1;
    Rosette_Border_ptr++;
  }
  else
    Rosette_Border_ptr->NextBorder = NULL;
}
```

Each XY pointer is linked to its respective line set. But wait a minute. Each line set consists of 11 segments and of course two end points. That makes 12 points, not 11. Note that at this point in the program, the variable "points" does not equal 11. It equals 12, exactly what we need. It became 12 when it exceeded the upper limit of the previous do loop. Tricky, eh? Each NextBorder is linked to the next Border structure, except for the last one which is assigned a NULL. The result of all this is a graphic structure that occupies better than half the memory of the previous example, yet produces the same drawing.

## Programming Challenge

Is there a way to program the rosette using a single Border structure? Yes there is, but it has an upper limit. That is, there is a limit in the point size of a rosette that can be programmed using a single Border structure. Can you find it?

Think about it, you have to design a single Border structure to draw a contiguous line that produces a complete rosette. I did not say that the line could not double up on itself, it can. But to produce the largest rosette possible, and remain within the upper limit of the Count member of the Border structure, it will have to draw in the most efficient way. Actually Rosette2A is a good hint on how to do it. Your solution must be programmed dynamically, to make most efficient use of memory, and it must be generalized to accept any POINT size up to its upper limit. You may use the simple method of defining a macro for the POINT size, as I did in this article. To test your program, simply change the macro and re-compile. If your programming skills are up to it, you may use command line arguments, or any other legitimate form of input, to enter different POINT sizes. But your solution must indeed produce a complete rosette for each POINT size up to its upper limit. It must also correctly handle the case when the point size is too high. Sorry, no system crashes are allowed.

*Good luck!*

The directory called Rosette2B contains one last example, which is equivalent to the one in Rosette2A except that its Count member has been reduced by 1. Many interesting, artistic patterns can be discovered by making small modifications to the types of example programs we have been using.

## Line Animation

Don't fall into the trap of thinking that DrawBorder( ) is useful only for static drawings. It can be used for animations as well. A good example is the Line_Animation program from two issues ago. You remember that one, don't you? It drew a line that left a trail as it moved around the screen. Perhaps you were not too impressed with it. After all, there is a similar one on the SAS/C examples disk that is 10 times faster. But wait, there is a big difference. Run the SAS/C example and you will see that as the trail is removed (by drawing the last line in the background color) it overwrites portions of the remainder of the trail. I know it's moving so fast you can hardly see it, but by pressing the right mouse button you will temporarily stop it and be able to see the effect I am talking about. My version does not do that. By relying on the fast drawing speed of DrawBorder( ), I was able to redraw all lines in the pattern every time the last one was removed. The effect is a much smoother animation.

I promised in that earlier issue that I would use the Line_Animation example in future articles to teach certain programming concepts, and I will, just as I did this issue with the rosette. It's just a little too soon yet for animation. In case you did not see the Line_Animation program, you will find a compiled version of it on disk this issue.

## Rendering Text

You have seen me slip in without explanation a few text instructions in all the examples of this series. It is now time to say a few words about that. As in the case of rendering line drawings, there are two ways to render text on the Amiga, a primitive way and a high-level way.

## PrimitiveText( ) Function

The Text( ) function is part of the Amiga's graphics library, which is automatically opened for you by the programming shell. Here is an example of its use:

```
SetAPen(my_rp, 1);
Move(my_rp, 0, 10);
Text(my_rp, "Help me, I'm stuck in here!", 17);
```

Initially you must select a pen register, and move the graphic cursor (drawing point) to the place on the screen where you want the text to appear. The Text( ) function itself requires three arguments. The first is the RastPort pointer, representing the window in which you want the text rendered. The second is the text itself, or a pointer to a NULL terminated length of memory containing it—more about that in a minute. The last argument is the number of characters in the string.

## Baseline of a Font

The word font refers to the characteristics of the letters used by the computer for displaying text on the screen. The Amiga has ten different fonts, with names like topaz, diamond, ruby, ... etc. They all come in a variety of sizes. In this article I will discuss only one, the Amiga's default font called TOPAZ_EIGHTY.

To place text at precise locations on the screen, you must have some knowledge about the font you are using. Specifically you must know the location of its baseline, the point in its graphic definition from which it will be rendered; its graphic reference point if you will. In previous examples I have used the number 10 as a vertical coordinate in the Move( ) function when I wanted to place text near the top of the screen. I chose that number to guarantee that the text would appear properly within the limits of the window, while at the same time not over-complicate the appearance of my code. But that was not quite correct. To place text properly, I should have used the font's baseline definition.

The height of a font is measured in scan lines starting from the top of the letters. The Amiga's default font, TOPAZ_EIGHTY, is defined in the header file <intuition/preferences.h> to be eight lines high, numbered from 0 to 7. In addition, its baseline is defined (in a different place) to be one line up from the bottom, or line number 6. But of course one should never go around using absolute numbers for such things. The user could easily change the default font, in which case the current font's baseline would be different and the positioning of the graphic cursor in your program wrong. The solution to all this is to read the baseline of the current font from the system. It is kept in the RastPort structure, in a member called TxBaseline. Below I show my above example modified to do this:

```
Move(my_rp, 0, my_rp->TxBaseline);
Text(my_rp, "Help me, I'm stuck in here!", 17);
```

Now I need never worry about the text accidentally over-shooting the upper edge of the window. The exact top line of the letters will always coincide with the exact top of the window. The user can change the system font and my program will comply by using the baseline of his selection.

## Building Your Strings

Remember that unlike BASIC, there are no real string variables in C. You have to build them up yourself by declaring character arrays and then copying text into them. Page L188 of your SAS/C documentation lists four different functions that you can use to do this, each one having its own advantages. I like to use stccpy( ) for a few reasons. First, its operation is always limited to some fixed number of characters that must be specified. That protects you from the disaster (possible system crash) of accidentally copying strings that are not properly NULL terminated. Second, when stccpy( ) encounters a NULL character in the source string, it fills the remainder of the destination string with NULLs, up to the length you specify. Third, it guarantees that the last character of the destination string is a NULL character, even if the source string is (accidentally) longer than the destination. Finally, it returns the length of the string copied, convenient for use in the Amiga's Text( ) rendering function. Here is an example:

```
#include <string.h>

#define STRING_LEN 128

char my_string[STRING_LEN];
int size;

size = stccpy(my_string, "Help me, I'm stuck in here!", STRING_LEN);
Move(my_rp, 0, my_rp->TxBaseline);
Text(my_rp, my_string, size-1);
```

I must subtract 1 from size because stccpy( ) reports length including the terminating NULL character. Of course you may have your own legitimate reasons for wanting to use another one of SAS/C's string copying functions, or even one of your own design.

## Using SprintF( )

You are probably wondering how to go about displaying numeric values into a graphic window. Intuition is not like a CLI window; you cannot use the normal Standard C printf( ) instruction. In fact, if your program executes a printf( ), its text goes, not to your programming shell window, but to the CLI (or AmigaDOS Shell) window from which you launched the program, what is called standard I/O.

A convenient function to use for getting numeric values into a graphics window is sprintf( ). It does everything printf( ) does, except that it sends the text to a previously declared char array, instead of stdio. It even returns the length of the formatted string produced. From there it can be easily used by Amiga's Text( ) function.

```
char my_string[STRING_LEN];
int size;

size = sprintf(my_string, "An approximation of PI is %f", 22.0/7.0);
Text(my_rp, my_string, size);
```

The sprintf( ) function allows you to use all the formatting conventions of Standard C to build strings that you can then render into any graphics window.

## TextAttr Structure

The programming shell window of these articles does not automatically support scrolling and carriage returns like a CLI or AmigaDOS Shell window. You have to keep track of where to display successive lines of text yourself. The minimum that you need to know is the height of the current font. That, after all, is what determines how many lines of text can appear on the screen, and how they should be spaced for best appearance. Font height can be read from the system by using the AskFont( ) function in conjunction with a TextAttr (text attribute) structure.

The TextAttr structure is defined in the <graphics/text.h> header file thus:

```
struct TextAttr
{
    STRPTR   ta_Name;    /* name of the font */
    UWORD    ta_YSize;   /* height of the font */
    UBYTE    ta_Style;   /* intrinsic font style */
    UBYTE    ta_Flags;   /* font preferences and flags */
};
```

AskFont( ) is a function in the Amiga's graphics library. It can be used to obtain the above information about the current font. It requires that you previously declare a TextAttr structure in which to report the information.

```
struct TextAttr default_Attr;
AskFont(my_rp, &default_Attr);
```

You can then read the height of the current font from that structure:

```
line_height = default_Attr.ta_YSize + 2;
```

I assign line_height to be two scan lines greater that the actual font in order that the text appears properly spaced on the screen. Two scan lines work out pretty well for TOPAZ-EIGHTY.

See the example Strings.c, in the directory Display_Text, on the magazine disk. It shows you how to use a variable, called line_number, to keep track of where to render successive lines of text to the screen. The example also contains the previously mentioned sprintf( ) function, reporting a floating point value to screen.

## Intuition's High Level Text Function

Just as there exists a high-level Intuition, line-drawing function, there also exists an equivalent text-rendering one, called PrintIText( ). Here is its operational theme:

1. Declare one or more IntuiText structures to contain the specifications of the text you want rendered: its color, position, font, ... etc. I like to call this the description structure.
2. Store the actual text you want rendered in a NULL terminated character array.
3. Link all structures and their respective text-arrays together.
4. Invoke the PrintIText( ) function.

Depending on exactly what kind of text you want rendered, there may be advantages to using PrintIText( ) over the more primitive Text( ).

## Some Background

All text consists of two parts, the letters themselves, called the foreground, and the background area over which they appear. For example, in TOPAZ_EIGHTY, each character is formed on an 8x8 pixel grid by illuminating certain pixels, while at the same time leaving certain others off. The pixels that are illuminated form the actual character and are called the foreground. Those that are left off are called the background.

The Amiga can render text in four different ways, described by its different DrawModes. The first is called JAM1, the same term that you saw earlier in DrawBorder( ). In JAM1, mode text is rendered such that the foreground of each character over-writes any previously drawn graphics or text, but the background does not. Thus the computer is rendering in one color only, the foreground color. A second mode, called JAM2, renders text in such a way that both foreground and background over-write previously rendered graphics or text. Thus the computer is drawing in two colors, the foreground color and the background color. A third mode, called COMPLEMENT, renders text such that all foreground pixels are replaced by their binary complement. The last mode, called INVERSVID, reverses the roles of the foreground and background colors.

## IntuiText Description Structure

The template definition for the IntuiText structure is in the file <intuition/intuition.h>. I reproduce it below without its comments:

```
struct IntuiText
{
    UBYTE FrontPen;
    UBYTE BackPen;
    UBYTE DrawMode;
    SHORT LeftEdge;
    SHORT TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
};
```

FrontPen is set to whatever color register you want for the foreground of the text. BackPen is set for the background. DrawMode is set to either JAM1, JAM2, COMPLEMENT, or INVERSVID. LeftEdge and TopEdge determine the position of the text in the window. TopEdge refers to the position of the top scan line of the text, not to the baseline as was the case with the Text( ) function. The ITextFont

member points to the structure which describes the font you want to use, or NULL if you want the system's default font. I will leave this NULL today. IText is a pointer to a NULL terminated contiguous section of memory containing the actual string that you want displayed. Finally NextText points to another IntuiText structure. You use this last member if your text consists of many strings, each one represented by a different IntuiText structure. Below I give a hierarchical drawing of a properly constructed IntuiText structure that displays a single string message at the very top left corner of the window. It uses the system's default font, color register 1 for foreground, and color register 0 for background. Its NextText member is assigned a NULL, meaning that I have only one string.

```
FrontPen    = 1
BackPen     = 0
DrawMode    = JAM2
LeftEdge    = 0
TopEdge     = 0
ITextFont   = NULL
IText———————————————————————→  Help me, I'm stuck in here! '\0
NextText    = NULL
```

The actual rendering is accomplished by the PrintIText( ) function:

```
PrintIText(my_rp, &My_IText, x, y);
```

... which requires four arguments, the RastPort pointer, the address of IntuiText description structure, and an offset coordinate pair that has a similar purpose to its equivalent in DrawBorder( ).

The example IText.c, in the directory Display_IText, demonstrates one way of using Intuition's PrintIText( ) function to display an entire screen of text. It opens an interlace screen, enough room for 50 lines in TOPAZ_EIGHTY. It then declares a 50-element array of IntuiText structures, one for each line on the screen, and a character array called my_string[50][80], enough memory space for 80 characters on each line. (79 actually, remember the '\0' character?) The IntuiText structures are then linked together and each one made to point to its own 80-character section of memory. The TopEdge member of each IntuiText structure is assigned a different vertical position, thus placing each line at its correct location on the screen.

```
for(i = 0; i < 50; i++)
{
    My_Page[i].FrontPen = 1;
    My_Page[i].BackPen  = 0;
    My_Page[i].DrawMode = JAM2;
    My_Page[i].LeftEdge = 0;
    My_Page[i].TopEdge  = i * 8;        /* position the line */
    My_Page[i].ITextFont = NULL;
    My_Page[i].IText     = &my_string[i][0];   /* point to text */

    if(i < 49)
        My_Page[i].NextText = &My_Page[i+1]; /* point to next line */
    else
        My_Page[i].NextText = NULL;      /* last one goes to NULL */
}
```

Text can be copied into the character array using stccpy( ). It's easy to keep track of things because the element numbers in the character array correspond to lines on the screen. Finally the entire structure is rendered using a single PrintIText( ) instruction. If you run the example, please do not try to read the text on the screen; it doesn't stay there long enough. The example is intended only to show you how PrintIText( ) can be used. It also shows how to clear the screen and the 50x80 character array in order to display new text.

For some simple text oriented programs this may be all the screen design that you need. It is easy to set up, uses high level language concepts, and renders text extremely fast. Naturally it should be structured into several functions before making it part of any application.

## Next Issue

This last example demonstrates a shortcoming of all my programming examples so far, that they remain on screen for only a predetermined period of time. I design them that way in order to keep them simple at these early stages. However, in the next issue I will present the missing link that will allow us to control programs using the mouse or keyboard, the Amiga's message system.

I will also present the Amiga's disk based font system and how you can use any one of them in your programs.

*Paul Catonguay loves to hear from readers! Please send your comments to Paul Castongauy, P.O. Box 505, Everett, MA 02149. Additionally, you can write to Paul c/o AC's TECH, P.O. Box 2140, Fall River, MA 02722-2140. We will forward your letters to Paul.*

☑

Recently, we were ordered by U.S. military officials to explain to their complete satisfaction just what a SuperSub is (as we all know, it's the best subscription deal around for Amiga users, since it includes both *Amazing Computing* and *AC's GUIDE*).

✩✩✩✩✩

Then, a prominent Congressman wired to ask us if we would testify before a top-secret subcommittee as to whether or not we can produce a single prototype SuperSub for less than $500 million (is this guy kidding? – a one-year SuperSub costs just $36 – and we can produce one for *anybody*!).

✩✩✩✩✩

Finally, a gentleman called us from Kennebunkport and told us to read his lips, but we told him we couldn't, because we don't have a picturephone.

And then he ordered a SuperSub.

---

# AC's SuperSub –
# It's Right For You!
# call 1-800-345-3360

---

## List of Advertisers

Please use a Reader service card to contact those advertisers who have sparked your interest. Advertisers want to hear from you. This is the best way they have of determining the Amiga community's interests and needs. Take a moment now to contact the companies with products you want to lean more about. And, if you decide to contact a advertiser directly, please tell them you saw them in

## AC's TECH /AMIGA

| Advertiser | Page | Reader Service Number |
|---|---|---|
| Central Coast Software | CIV | 104 |
| Delph Noetic Systems | 31 | 101 |
| Great Valley Products | CII | 105 |
| Great Valley Products | CII | 105 |
| Memory Management | 93 | 103 |
| Radical Eye Software | 63 | 102 |

---

## Do you have a story idea for AC's TECH?

## We want to know!

**(Call or write the Editor)**

# The Development of a Ray Tracer

## Part Two: The Implementation

### by Bruno Costa

In the first part of this article, some of the essential ray-tracing concepts were presented, including a description of a simple illumination model. This second part is dedicated to show the practical usage of that theory, with many commented excerpts of code from a real ray tracer (called "ray"), that is included on disk with the full source code in C. Some references will be made to the first part of the article, and, although not strictly necessary, it is recommended that you try to understand that part before reading on. If you are not specifically interested in ray tracing, it may be worthwhile to examine just the object orientation sections.

## The General Structure

The outline of the basic ray-tracing algorithm presented in the previous issue revealed how simple it can be. It was mentioned that what distinguishes a good ray tracer are the special features it has, besides the almost standard ray-tracing kernel. This is one of the reasons why ray tracers are not merely a few hundred lines long. Another important factor is the code necessary for each kind of object, the various data types (like colors and vectors) and the output device drivers.

Since ray tracing needs to know some features of each kind of object it is supposed to render—how to calculate an interception of the object with a ray, for instance—additional code is needed to support each class of objects. These classes of objects are often called primitives, because they can be combined to produce much more complex objects.

Some libraries and routines are used by all other parts of the program to help in manipulating 3D vectors, colors, lists of objects, parse command line options and handle errors—they comprise the support code. Also important are the routines that communicate with the output devices (like an Intuition HAM screen, an IFF file, or a 24-bit frame buffer), which guarantee some device limitations or restrictions and also perform any necessary steps to control the hardware or software involved.

With these classifications in mind, a good organization for a ray tracer could be the one given by Figure 1. The user sees simply a front-end, called "ray," that controls the ray-tracing kernel. To do the real ray tracing, the kernel needs to call each primitive to determine some of its features, and the output device driver to show or store the resulting picture. The various small packages in the support code are called by all other parts of the program, specially to manipulate some common data types.

An important characteristic, depicted in Figure 1, is the way the modules communicate. Between each pair of communicating modules (or groups) is an appropriate interface, for instance, the user communicates with the program front-end through the user interface. Each interface can usually be described by a well-defined, limited set of function calls (with applicable parameters) and data types. This is the case of all the interfaces in Figure 1, except for the user interface, that is obviously described instead by a very complex collection of commands, options, procedures, and even the input and output devices involved.

A question that is naturally raised is how does one extend the abilities of this ray tracer, particularly how does one add a new primitive? This is a very important question indeed, since after the ray tracer is relatively stable, the most frequent modifications made on it will be to add support for new primitives. It is desirable that this operation is simple enough that even someone who is not completely acquainted with the program is able to do it, ideally minimizing or even removing the need to modify the program itself. Well, this is the subject of the next section.

## Object Orientation Prelude

If you have been programming for a while—in fact, even if you haven't—certainly the words "object oriented" mean something to you (maybe "yes, they mean something I don't understand"!). "Object oriented" is definitely one of the most fashionable terms in computer science nowadays—we have object-oriented languages, compilers, operating systems, databases, user interfaces, and even ray tracers. If you say your program or product is "object oriented," people look at you differently, respectfully (even if the word does not express much to them). Well, what does all this object orientation stuff mean, anyway? A complete answer is not easy, but what most people imply when they say something is object oriented, is that it is well structured, divided in modules, and encapsulated (didn't help much, did it?).

As you are probably aware, modularization and structures are two of the classical tools that a programmer has in order to make his program more manageable. By dividing the program in small routines, each with its own well defined purpose, one can write big (and also small) programs that tend to become easier to fix and maintain. If you have ever written a medium-sized program in the early microcomputer BASIC's—either C-64, Atari, TRS-80 or Apple—you surely know what I am talking about. Most modern languages support an additional level of modularization, by allowing you to group related functions in a single module. The parts of a module that are visible outside make up the module interface, i.e., the entry points that can be called from outside. Structures allow you to collect related data in a single place, building more complex data types from simpler ones.

Figure One — How the modules communicate
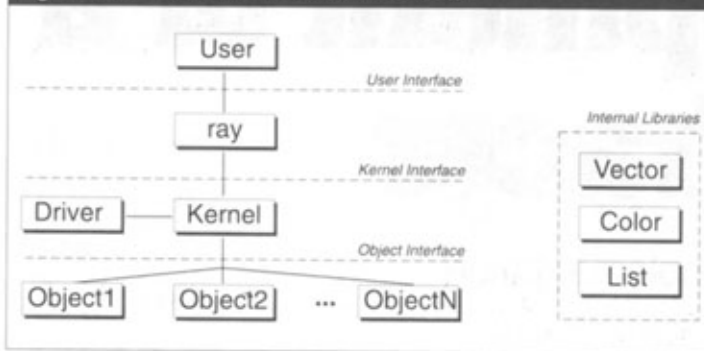
We can take the C language as a convenient example. In C, functions and variables can be grouped in files, and any functions or global variables preceded by the keyword static are not visible outside of the source file they are in. Usually, the prototypes of the externally visible functions (defining the name of the function, its return value, and the parameter types and their order), as well as the data types handled by the functions in that module are defined in a header file (name ending in ".h").

Typically, a structure is used to define a complex data type, that will be manipulated using a set of functions. You can think of a list, defined by a simple structure and manipulated by the routines addtail, addhead, removehead and removetail, for instance. So, it is relatively clear that there exists a relationship between the data type and the functions able to manipulate it. It would be nice to have an extra level of grouping that allowed us to define a data type completely, in a single place, including both the data structure and the functions that understand and manipulate it. Ideally, only the functions in the definition of the type would be able to read or modify the data structure fields directly—everyone else would have to call these functions to do it. This would allow the real data structure organization to change at will—as long as the manipulation functions are modified accordingly—without the callers ever noticing it. Well, programs that use this extra level of modularization are usually called object-oriented programs.

I have purposefully avoided the use of the object-orientation jargon, so that uninitiated readers could understand the preceding discussion. The data types defined by a module with an internal data structure and some manipulation functions are generally called objects. The manipulation functions are called methods. The methods and the data type define in fact an object class, and each individual object belonging to that class is called an object instance. For example, the object class BALL might define objects that can, according to some kind of semantics, roll, stop, or kick (the three methods that can be applied to BALLS). Both a small red ball and a big blue ball are distinct instances of BALL, and both can be rolled, stopped, or kicked using the same methods defined by the BALL class.

There are additional concepts that are important to achieve the exceptional level of encapsulation that truly object-oriented programs have, most of which are tricky to implement using a language that is not object oriented (like C). Probably the most important of them is inheritance, the ability of one class of objects to inherit properties of another class. This allows one class that is a particularization of another to use all the methods that were defined for the more general one. A good example is a class SHAPE, with the methods rotate, translate, and scale, and sub-

classes that inherit the properties of a SHAPE: CIRCLE and SQUARE. Thus, besides using the methods particular to the CIRCLE class (setradius, for instance), it is also possible to apply a translation, for instance, to an object of the class CIRCLE—after all, CIRCLEs are SHAPEs, aren't they?

### Practical Object Orientation

As you can notice by the previous example, object-oriented techniques are particularly suitable to describe graphical objects, and that is precisely what a ray tracer needs to do. A ray tracer manipulates graphic primitives, like spheres and polygons, to which standard methods like intercept and normal are applied to obtain the necessary information to render a picture. Each kind of primitive has some very specific attributes, mainly related to its geometric shape, besides those that are common to all primitives (like color and surface properties). This is a typical case of inheritance: each kind of primitive is a class that inherits the attributes and methods of a general OBJECT class.

You should be convinced by now that using object-oriented techniques to write a ray tracer seems like a good idea. A fully object-oriented language would be the natural way to go, but is it really required? Truly object-oriented languages are sometimes too different from what you are used to, introduce some inefficiencies and are not widely available. Hmmm ... would it be possible to implement some of the nice object-oriented ideas using the language of choice of nine in 10 Amiga programmers? With some restrictions, the answer is yes.

Since C is not an object-oriented language, there is a slight overhead in using object-oriented techniques in a program. Due to this small inconvenience, the only place where object orientation is used in "ray" is in the object interface, where it is successful enough to make this overhead negligible—you will notice why. So, "ray" is not a fully object-oriented program, just like C is not really an object-oriented language.

In "ray" there is a general class, called Object, that allows every kind of primitive to be treated similarly. The ray tracer kernel, for instance, deals only with Objects—it simply does not know what kind of Object it is really rendering. This class supports all the methods necessary for a primitive to be ray traced, although each particular kind extends an Object with attributes particular to its geometry. An Object in "ray" is defined in object.h as:

```
typedef struct {
  ListNode o_node;
  objtype o_type;
  Color o_color;
  Illum o_illum;
  void *o_data;
} Object;
```

where objtype is an enumerated type that lists all the possible subclasses of Object:

```
typedef enum {
  OBJ_SPHERE,
  OBJ_SKY,
  OBJ_PLANE,
  OBJ_NUMBER /* number of defined objects */
} objtype;
```

An Object thus contains: a node sub-structure to allow the object to be linked to a list; an identifier of which type of object this instance is; the color of this object; the illumination characteristics of the surface of this object; and finally, a generic pointer to data particular to the type of this object. You should notice that fields in this structure, like color and illumination characteristics, are attributes that every primitive, regardless of its particular type,

will inherit. The extension to the Object class is made through the o_data generic pointer, to which each primitive can link any special data it might need. For instance, in sphere.c the Sphere type is defined as (basic types like real and Point are defined in global.h):

```
typedef struct {
  Point s_center;
  real s_radius;
} Sphere;
```

When a sphere is created, an Object and a Sphere structure are allocated and initialized, and a pointer to the Sphere is placed in the o_data field of the Object. No one—except the methods defined in sphere.c - knows what is stored in the o_data field, i.e., no one knows that it points to a structure made up by a Point and a real. This is information hiding: if, for any reasons, it is necessary to change the Sphere structure, no one—except the methods defined in sphere.c—will need to change.

This is how the attributes of the Object class and its subclasses are stored, and how inheritance works in "ray." More important than that is how it is possible to have a consistent, essentially primitive-independent way of accessing the crucial characteristics of an object to be ray traced.

## Attribute Manipulation

When an object is created, the fields in the Object structure, and specially the data pointed to by the o_data field, are initialized with default values. Since the value of these fields is what effectively distinguishes two instances of a given class, there must be a way to modify and retrieve them. Instances of the Sphere class, for instance, are always created with center at the origin, a radius of 1 and a diffuse white, non-reflective, non-refractive surface.

It was mentioned that only the methods in the subclass implementation know what is stored in the o_data field, and so, only a method in the subclass can possibly modify these attributes. Also, since there is no limit on the number of primitives implemented nor in the number of attributes they support, the process used to modify attributes must not depend on these.

We want the user to be able to modify the attributes of an object in a controlled way, still keeping him unaware of what is effectively stored in the structures. If, for any reasons, we need to radically change the format or kind of data that is stored in the object, we would like to be able to have the opportunity to map user requests in the old format to the new one, and also hide the internal fields that are meaningless to the user.

The obvious solution that meets most of the above criteria is a pair of methods, *attrset* and *attrget*, that are used to set and get the values of attributes. The attribute that is to be read or modified must be identified by a code, and this code is given by the enumerated type attribute, defined in attributes.h as:

```
typedef enum {
  ATTR_END,
#include "common.atr"
  ATTR_CUSTOM,
#include "sphere.atr"
#include "plane.atr"
#include "sky.atr"
  ATTR_DUMMY
} attribute;
```

As is relatively obvious, this definition is not self-contained, since most of it is included from external files. This is done to make the addition of new primitives easier, reducing the number of modifications required in attributes.h to one. A file like sphere.atr looks like (stuff deleted):

```
ATTR_LIST_BEGIN

  SPHERE_CENTER = ATTR_CUSTOM,
  SPHERE_RADIUS,
  SPHERE_NUMATTR

ATTR_LIST_END
```

where ATTR_LIST_BEGIN and ATTR_LIST_END are macros defined in attributes.h. The attributes common to all primitives are defined in common.atr.

Note that the first custom attribute of each primitive is initialized to be ATTR_CUSTOM. This technique results in a unique integer code for each attribute applicable to a given primitive, but custom attribute identifiers of different kinds of primitive may (and in fact they will) be equal, as is shown in Figure 2. Note also that the last attribute of each primitive is usually primitive_NUMATTR, and this will be exactly the number of useful attributes for that primitive. In a Sphere, for instance, there are six attributes that can be used (SPHERE_NUMATTR = 6): ATTR_END, ATTR_COLOR, ATTR_TEXTURE, ATTR_ILLUM, SPHERE_CENTER and SPHERE_RADIUS.

A question that is still remaining is how we will be able to define a function to set an attribute of an object, if the type of the attribute is not known. A good solution is to use the ANSI C variable arguments-passing facility.

## Variable Arguments

One of the best known functions in the C standard library is printf(). There is something special about printf, besides the frequency that it is used in normal C programs: the fact that a variable number of arguments can be passed to it. If you have ever tried to write a printf-like function, you might have stumbled on a shortcoming of many programming languages: most of them do not give you the means to write functions or create modules that work just like the standard library ones do. There is simply no way, in standard Pascal (is there really a standard?), for instance, to write a function that can receive a variable number of arguments (like Write() and Read() do) or an argument of any type (like New() does). This kind of contradiction prevents the use of the language in question to write its own standard library (forcing the use of another language, usually assembly), reduces the portability of the language implementation and frustrates programmers trying to write code that looks like the standard they are used to.

One of the nicest—and also one of the least used—features standardized by the ANSI C committee is the variable argument-passing feature. It allows a routine to receive any number of

**Figure Two**   Attributes

| Code | | | |
|---|---|---|---|
| 0 | ATTR_END | | |
| 1 | ATTR_COLOR | | |
| 2 | ATTR_TEXTURE | common.atr | |
| 3 | ATTR_ILLUM | | |
| 4 | ATTR_CUSTOM | sphere.atr | plane.atr |
| 5 | | SPHERE_CENTER | PLANE_ORIGIN |
| 6 | | SPHERE_RADIUS | PLANE_NORMAL |
| | | SPHERE_NUMATTR | PLANE_NUMATTR |
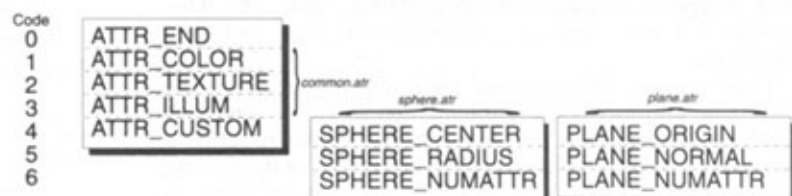
parameters of any type, just like printf does, and to retrieve the parameters from the stack in a standard way, independent of the particular machine architecture. This feature is implemented as a single type and three macros, defined in the include file stdarg.h:

```
void va_start (va_list args, lastfixed)
void va_end (va_list args)
type va_arg (va_list args, type)
```

To use it, you must declare a variable of type va_list, that is used to keep track of which parameter we are currently in. It is initialized by a call to va_start, that receives the last fixed parameter in the function. A call to va_end must be made after the arguments have been used, in the same function that called the corresponding va_start. To retrieve the current parameter and advance to the next, va_arg should be used. It receives the type of the parameter expected, and returns its value.

The best way to understand this feature is with a simple example:

```
/*
 *   maxint - returns the maximum of a
 * series of positive integer arguments
 * terminated by 0.
 */
int maxint (int i, ...)
{
 va_list args;
 int max = 0;

 va_start (args, i);
 while (i)
 {
  if (max < i)
   max = i;
  i = va_arg (args, int);
 }
 va_end (args);

 return max;
}
```

There must be at least one fixed argument to initialize the va_list, and the argument list in the prototype of the function must be terminated by "..." to indicate that there is an unknown number of arguments following. It is very important to keep in mind that there is no way to know how many arguments, and of which type, were really passed to your function, so you must provide an alternate way to get this information, either implicit or explicit. In the function above, the type of the arguments is implicitly assumed to be integer, but the number of them must be explicitly marked by the user by terminating the argument list with a zero:

```
maxint (7, 98, 37, 0);    /* OK. */
maxint (19, 1.0, 2, 0);   /* WRONG! */
maxint (5, 2, 9);         /* WRONG! */
```

In printf(), the number and type of the arguments must be explicitly listed in the format string, telling the function the type of each parameter, and the order they are received. The only problem with this explicit user cooperation is that sometimes he/she might write a format string that is incompatible with the number or type of the arguments really passed, and ... POOF!— the program doesn't work any more, usually with the most bizarre effects.

These strange reactions from the computer are due to the way variable arguments are implemented in usual computer architectures. Arguments are passed in the stack—roughly, a memory area that grows either toward increasing or decreasing addresses—one next to the other. A va_list variable is simply a pointer to a place in that area, which va_arg moves by an amount that is the size of the argument retrieved by it, so that it will be always pointing to the next argument. If, for any reason, the type of the argument passed to va_arg is wrong or the arguments have just ended (and the function doesn't know), va_list will be pointing to the wrong place, usually an address that stores just garbage, and the values returned from subsequent calls to va_arg will be plain nonsense. You can imagine the rest.

You might have already guessed how variable arguments can be used to implement the attrset and attrget methods. To set an attribute, for instance, the method has to know which object is to be modified, which attribute will be changed, and the new value of the attribute. Only the type of the latter can vary, depending on which attribute is being changed. Thus, somehow there must be a way to associate each attribute identifier to its type, so that the method will know which type to remove from the stack. This connection is made through an array of types indexed by the attribute identifier, that will be present in each primitive. The possible types, defined in attributes.h, are:

```
typedef enum {
  TYPE_INT,    /* integer argument */
  TYPE_REAL,   /* real argument    */
  TYPE_PTR,    /* pointer (address) */
  TYPE_VOID    /* unused argument  */
} attrtype;
```

The type array for the sphere is (from sphere.c):

```
internal attrtype stypes[SPHERE_NUMATTR] = {
  TYPE_VOID, /* Always begin with TYPE_VOID */

/*
 * Common attributes
 */
  TYPE_PTR,     /* ATTR_COLOR   (Color *)     */
  TYPE_PTR,     /* ATTR_TEXTURE (Texture *)   */
  TYPE_PTR,     /* ATTR_ILLUM   (Illum *)     */

/*
 * Private attributes
 */
  TYPE_PTR,  /* SPHERE_CENTER (Point *)   */
  TYPE_REAL  /* SPHERE_RADIUS            */
};
```

You should note that there are six items in this array, one for each of the six attributes that are valid for Spheres, and that they are ordered just like in Figure 2.

Since we are using variable arguments, the attrset and attrget methods can be designed so that they set or get more than one attribute at a time. This is easily done by defining the functions according to the following prototypes:

```
int attrset (Object *obj, ...)
int attrget (Object *obj, ...)
```

Following the object to be manipulated there is a list of attributes and their values, terminated by an ATTR_END, for example:

```
Color white = {1.0, 1.0, 1.0};
attrset (sph,
     SPHERE_RADIUS, 2.5,
     ATTR_COLOR, &white,
     ATTR_END);
```

In the case of the attrget method, the arguments are pointers to the variables in which the value of the attributes will be stored:

```
Color color;
real r;
attrget (sph,
     SPHERE_RADIUS, &r,
     ATTR_COLOR, &color,
     ATTR_END);
```

Looking at the array with the types for the sphere, you can determine the type of the value that goes after each one of the attributes, e.g., after a SPHERE_RADIUS there must be a real, an ATTR_COLOR must be followed by a Color pointer, and so on. Don't forget that in the case of attrget all values are pointers, in such a way that values can be returned in the area pointed to by them. To better understand this, you may associate attrset and attrget to printf and scanf, respectively.

## Methodology

As already mentioned, the two fundamental methods that the Object class must implement are intercept and normal. Both of them depend entirely on the kind of object in question: the Object class simply doesn't know how to calculate an interception or a normal without knowing that the object is in fact a sphere or a plane, for instance, and this will be the case of most methods in this ray tracer (well, currently all of them). So, to implement this, the intercept method in the Object class could have been written as something like:

```
real intercept (Object *o, Line *l)
{
  if (o->o_type == SPHERE)
    return sph_intercept (o, l);
  else if (o->o_type == PLANE)
    return pl_intercept (o, l);
  ...
  else
    error ("Unknown object type!");
}
```

There are, however, better ways to implement the above, that will group all the information related to each kind of primitive in a single place, instead of scattering it through many parts of the program. Each subclass declares all the properties it supports in a structure that is the only thing visible outside of that subclass implementation. This structure is defined in properties.h as:

```
typedef struct {
  Object *(*op_create) (void);
  void (*op_destroy) (Object *);
  void (*op_normal) (Object *, Vector *, Point *);
  void (*op_colorat) (Object *, Color *, Point *);
  real (*op_intercept) (Object *, Line *);
  int (*op_attrset) (Object *, ...);
  int (*op_attrget) (Object *, ...);
  attrtype *op_attrtype;
} Properties;
```

The above structure declaration is the kind of thing that really shocks C neophytes—well, admittedly, even seasoned programmers need to take a deep breath before trying to write such a thing. A rough translation: each of the first seven lines declare a pointer to a function, to C code that can be executed. The fifth line, for instance, declares a field named op_intercept that points to a function that accepts two parameters, an Object pointer and a Line pointer, and returns a real. A call to one of these functions using the pointer looks like:

```
extern Properties prop;   /* initialized elsewhere */
Object *redsphere;        /* (likewise)          */
Line *ray;                /* (likewise)          */
real t;

t = (*prop->op_intercept) (redsphere, ray);
```

In essence, each of the function pointers in the Properties structure is a method, and each primitive declares a Properties structure where each field is filled with the appropriated function that implements that method for that primitive. In the sphere class, for instance, there is a Properties structure named

sphere_prop that is globally visible, and it contains pointers to the functions that implement those methods, which are not directly visible outside. From sphere.c (stuff deleted):

```
Properties sphere_prop = {
  screate,
  sdestroy,
  snormal,
  scolorat,
  sintercept,
  sattrset,
  sattrget,
  stypes
};

internal Object *screate (void)
{
  ...
}

internal void sdestroy (Object *obj)
{
  ...
}

. /* some methods skipped here */

internal int sattrget (Object *obj, ...)
{
  ...
}
```

In this way, all the information on the methods for each primitive are grouped together, and easily accessible by the Object class. Better yet, the Object class can have an array of Properties directly indexed by the type of the object, like in object.c:

```
extern Properties sphere_prop;
extern Properties sky_prop;
extern Properties plane_prop;

/*
 * IMPORTANT: same order as the
 * objtype enumeration!
 */
static Properties *properties[OBJ_NUMBER] = {
  &sphere_prop,  /* OBJ_SPHERE  */
  &sky_prop,     /* OBJ_SKY     */
  &plane_prop    /* OBJ_PLANE   */
};
```

This allows the intercept method of the Object class shown before to be rewritten in a more concise way, that is also much more efficient and independent of the currently implemented primitive types:

```
real intercept (Object *o, Line *l)
{
  return (*properties[o->o_type]->op_intercept) (o, l);
}
```

Obviously, the methods pointed to by the fields in the Properties structure are in the code of the primitive in question, so they know everything about it, including its geometry and what kind of information is stored in the o_data field.

Let's get back to the definition of the Properties structure. There is one last field that was not yet mentioned, op_attrtype. It is simply a pointer to an array of attribute types, exactly like the one shown in the Variable Arguments section. You might still be wondering what each of those methods in the structure does, and why they are there. The methods are:

| op_create | used to create an instance of a primitive of a particular type (allocates memory and initializes it). |
|---|---|
| op_destroy | simply destroys a given instance of a primitive, freeing the memory used. |
| op_normal | returns the normal of a primitive at a given point (assumed to lie on the surface of the primitive). |
| op_color | returns the color of an object at a given point (also assumed to lie on the surface). |
| op_intercept | returns the interception of a given line, in parametric form, with an object. |
| op_attrset | sets each of the requested attributes of the object to the corresponding value given. |
| op_attrget | returns the value of each of the requested attributes in the corresponding value pointer. |

Each of the above has a corresponding method implemented in the Object class, like the intercept method already shown. If you look carefully in object.c, you will notice that all of the methods implemented there simply forward the requests to the appropriate subclasses that know how to handle them. You might be thinking what may be gained by introducing functions that do essentially nothing. Look at the following piece of code:

```
Object *sphere, *plane;
extern Line *ray;     /* initialized elsewhere */
real tp, ts;

sphere = create (OBJ_SPHERE);
plane = create (OBJ_PLANE);

tp = intercept (plane, ray);
ts = intercept (sphere, ray);
if (tp < ts)
 printf ("plane is nearer");
else
 printf ("sphere is nearer");

destroy (plane);
destroy (sphere);
```

It may look surprising, but the code above does work in "ray," actually as a result of the object orientation techniques just described.

### The Ray Tracer Kernel

The ray tracer itself is quite simple, and it is completely contained in the file ray.c. It is basically the algorithm presented in the last issue, with additional functions to compute the Whitted illumination model (local()), the reflection contribution (reflect ()), the transmission contribution (transmit ()) and the shadows (obscured ()). A single ray is recursively traced by raytrace() and illuminate(), and raydispatch() traces all the rays in a picture, sending the results to the device driver through the routines picture.c.

You will notice that the main() function in ray.c calls two world manipulation functions, wbuild() and wdestroy(). These functions are in input.c and are used to build a world (containing all the features of the scene, including the objects and the lamps) from a ray scene description input file. If you look carefully at the code in input.c, you might observe that it depends not only on the types of primitives currently implemented, but also on the attributes they support. But wait! Isn't this exactly the kind of thing we were trying to avoid by using object orientation? Er ... yes, but you'll see why.

In this case, there were essentially three alternatives. The first is the one currently implemented, where the knowledge of the primitive types and their attributes is not completely contained in the primitive implementation, making the addition of other primitives somewhat harder.

The second one would be to create an additional method for every primitive, that would read its parameters from a given file. In this way, the wbuild() routine, instead of reading each of the values and passing them to the primitive, would send the entire line to be processed by the primitive. Unfortunately, this method would make the primitives dependent on the kind of modeling interface in use (scene description file), and would make it impractical to change the program to run under an Intuition interface, for instance.

The third one would be to simply ignore input.c and use the routines create(), destroy(), attrset(), attrget() and raytrace() to model and render scenes directly. This is by far the cleaner and simpler solution, but it introduces the drawback of needing recompilation every time the scene is changed. Also, one executable is needed for each scene, and, unless "ray" is made into a shared library, the code for the ray tracer and the primitives will be duplicated in each executable.

The initial idea was to support the programmatic interface to model scenes, but it became somewhat awkward to use. The scene description file was then created to provide an easier and faster way to model scenes. With few changes, it is still possible to use the programmatic interface, simply by discarding input.c and writing two substitute routines: wbuild() to create all the necessary objects, and wdestroy() to destroy them. It may also be possible, if carefully planned, to use the second solution in a way that is less dependent on the kind of interface, maybe creating a method that maps a string to an integer ID (the string "ATTR_COLOR" would be mapped to the number ATTR_COLOR).

### Food for Thought

The alternatives above and the idea of changing the interface used to access "ray" create interesting possibilities of an interactive version of the program or a ray tracing shared library. Although textures are not implemented, the colorat() routine was designed specifically to support solid textures easily. Refraction is not implemented, but the rest of the program is prepared to handle it, and the basic theory was explained in the previous issue. There are various illumination models out there, which create truly realistic effects. Ray tracing time can be substantially reduced by many existing acceleration techniques. New primitives can be added easily (step-by-step procedure explained in template.c), and this fact alone can be the source of hours of entertainment (and debugging!).

In brief, there are lots of things to be done. If you have the time and the will to explore this package, I hope that you learn about computer graphics and programming in general as much as I did. If you want to delve into more advanced algorithms, try the bibliography listed in the last issue, and if you have any problems with this implementation, you may try to reach me on Internet as bruno@brlncc.bitnet or through AC's TECH. I hope you have fun!

☑

# Low-Level Disk Access Made Easy!

## by Dan Babcock

Floppy drive access is usually performed via the trackdisk.device module of the operating system. Trackdisk provides the same consistent, high-level interface as other device drivers in the system, and has been optimized internally for speed. Given that, there is usually no reason not to use trackdisk when multitasking normally. Special non-multitasking applications must, however, resort to programming the hardware directly. The purpose of this article is to present a set of easy-to-use routines for performing floppy access without the aid of the operating system.

### Using the Routines

The disk I/O package (Listing 2) consists of six major routines: Read, Write, MotorOff, Inquire, SeekZero, and RestoreHeadPosition. Read and Write perform the actual disk operations. MotorOff turns off all drive motors (Read and Write automatically turn them on), and deselects all drives. Inquire tells you what drives are present in the system. SeekZero sends all drives to track zero, placing the drives at a known position so that seek operations are performed correctly, while recording the previous track location of the drives in the DISK_OldTrack field of the DISK_DataArea global data area. SeekZero also does the favor of setting up the relevant CIA data direction registers. RestoreHeadPosition steps the heads to the DISK_OldTrack tracks, previously recorded by SeekZero. The register inputs and outputs of these routines are summarized below. Note that there are absolutely no restrictions on any of the parameters; for example, the buffer pointer may point to an odd address in fast RAM.

### Read

*Inputs*
> D0.L - length (bytes)
> D1.L - offset from start of disk (bytes)
> D2.L - drive (0-3)
> A0.L - pointer to user buffer

*Output*
> D7.L - error code (zero if no error)

### Write

*Inputs*
> D0.L - length (bytes)
> D1.L - offset from start of disk (bytes)
> D2.L - drive (0-3)
> A0.L - pointer to user buffer

NOTE: The write routine offers a verify option that may be enabled by changing the VERIFYFLAG constant near the beginning of the listing.

*Output*
> D7.L - error code (zero if no error)

### Inquire

Output
> D0.L - drive map
> - example: 5 (0101) means drive 0 and 2 are present.

### Other Routines—

> MotorOff:
> SeekZero:
> RestoreHeadPosition:

### No Inputs or Outputs

Some set-up work is required before these routines may be called. The DISK_RawBuffer and DISK_DecodedBuffer fields of DISK_DataArea must be set to point to free memory areas of size 14,716 bytes and 5,632 bytes, respectively. The DISK_CurrentTrack entries must agree with the physical position of each drive; you must call SeekZero or otherwise assure that that's the case. You should normally also call Inquire to find out what drives are installed. After completing this set-up, Read and Write may be called freely to perform the actual job. Read and Write make few assumptions about the state of the machine, and take care not to drastically alter it. This permits use in a wide variety of "hostile" environments.

### Drive Specifications

Almost every veteran Amiga user has experienced the frustration of a program (usually a game) absolutely refusing to load. When it comes to disk drives, playing fast and loose with the manufacturer's specifications almost assures that your program will

fail to work properly on someone's machine. For your convenience, the most critical specifications are listed in Table 1. In addition to head stepping considerations, take into account variations in drive speed, which appear at the programmer level as a variation in track length. Write a gap of 1,660 bytes to ensure that the track is completely erased, and plan for the gap to be that large when reading. That translates to a read size of 14,716 bytes and a write size of 13,628 bytes. The read size is one sector (1088 raw bytes) larger to ensure that 11 complete sectors are read. These numbers are used by trackdisk (and these routines), and your application will work just as reliably as trackdisk if you use them too.

### Table One

| | |
|---|---|
| Step rate | -3ms* |
| Settle time | 15ms |
| Post-write delay | 2ms |
| Side select delay | 1ms |

*4ms when looking for track zero

## Inside the Routines -

Some key facts about the internal workings of the disk routines will be mentioned here. First, the blitter is not used for encoding or decoding. Using the blitter would speed things up, but would make the routines less general purpose. As it is, decoding a track takes about 16 milliseconds, and encoding a track takes about 68 milliseconds (on a 68000/68010 Amiga). The decoding time is hardly noticeable, but the encoding time makes writing seem a bit sluggish. Secondly, the CIA timers are not used to implement the necessary delays; rather, the delay is based on counting a certain number of horizontal scan lines, at approximately 63 microseconds each. This approach leaves the timers completely free for other concurrent uses. As you can see, the design of the routines facilitates a simple drop-in inclusion into almost any program, requiring very little external support.

## Taking Over Gracefully ——

For testing these routines in a normal (multitasking) environment, it is desireable to take over floppy control functions from the operating system. The OS-friendly method of doing so is to call DR_GETUNIT in disk.resource. Once permission has been obtained, our routines may directly access the disk hardware without fear of interference from the operating system. Of course, normal multitasking rules must still be obeyed. Listing 1 shows a program that demonstrates calling DR_GETUNIT before making use of the low level disk I/O routines, and cleaning up afterwards.

## This is Not the End -

The disk routines are not presented with the intent to be the "last word" on disk routines. Rather, they are a very useful starting point; you don't have to reinvent the wheel. You are encouraged to make your own special improvements, of course and, we hope, share the result with everyone. I've already incorporated the disk routines successfully into VBRMon (see last issue's article), and no doubt you will find your own applications.

## Recommended Reading -

*The Hardware Reference Manual* is the official guide to programming the hardware, and should always be consulted first. The standard Amiga disk format is documented in the *Rom Kernel Manual: Libraries and Devices*. I found Abacus's *Amiga Disk Drives Inside and Out* to be quite helpful, though it contains many errors. Lastly, Randell Jesup's "More on Low-Level Disk Access," published in Commodore's AmigaMail, contains many very helpful tips.

## About the Author

Dan Babcock is an electrical engineering major at Pennsylvania State University and an avid assembly programmer. Contact him via Internet as d6b@ecl.psu.edu.

### Listing One    disk.s

```
;disk.s - Package for reading/writing the standard trackdisk format
;         without the OS

;Copyright (c) 1991 by Dan Babcock

;Access the routines as follows:

;For reading:
;Routine name: Read
;D0.L - length (bytes)
;D1.L - offset (bytes)
;D2.L - drive (0-3)
;A0.L - buffer

;For writing:
;Routine name: Write
;Same parameters as above.

;NOTE: Error code returned in D7.L - 0 if ok, else error

;General errors
DISK_BadParameter       equ         1

;Read errors
DISK_NoSync             equ         2
DISK_BadHeader          equ         3
DISK_BadData            equ         4

;Write errors
DISK_WriteProtected     equ         5
DISK_VerifyError        equ         6

;Additional routines:
;MotorOff - Turn off all drive motors
;Inquire  - Find out what drives are in the system
;           Returns drive map in D0
;SeekZero - Send all drives to track zero
;RestoreHeadPosition - step drives to previous (before SeekZero) position

;————— User-settable parameters —————
VERIFYON    equ         1           ;0=no verify
;

            IFND        dskpt
dskpt       equ         dskpth
            ENDC

;Global data structure

            clrso
DISK_CurrentTrack       so.b        4
DISK_RawBuffer          so.l        1
DISK_DecodedBuffer      so.l        1
DISK_OldTrack           so.b        4
DISK_Size               soval

;           aseg
;           objfile     'disk.binary'
```

```
;       bra.w       Read        ;offset 0
;       bra.w       Write       ;offset 4
;       bra.w       MotorOff    ;offset 8
;       bra.w       Inquire     ;offset 12
;       bra.w       SeekZero    ;offset 16
;       bra.w       RestoreHeadPosition   ;offset 20

DISK_DataArea:
        dcb.b       4,0         ;CurrentTrack - offset 24
        dc.l        0           ;RawBuffer - offset 28
        dc.l        0           ;DecodedBuffer - offset 32
        dcb.b       4,-1        ;OldTrack - offset 36

;Note: This is a public value, but it is NOT meant to be changed!!!
DISK_RawBufSize     equ         14716

ciabprb     equ     $bfd100
ciaapra     equ     $f01
ciabddrb    equ     $200
ciaaddra    equ     $1101

  comment |

****************************************************************
Important disk parameters, summary:

Step rate: 3ms. 4ms when looking for track zero.
Settle time: 15ms
Post-write delay: 2ms
Side select delay: 1ms
****************************************************************
|

Read:
        movem.l     d0-d5/a0/a1/a4-a6,-(sp)
        moveq       #0,d7
        lea         DISK_DataArea(pc),a5
        move.l      #_custom,a6
        move.l      #ciabprb,a4

        move.l      d1,d3
        add.l       d0,d3
        cmp.l       #901120,d3
        bhi.s       .Error      ;go if >

        bsr.b       SelectDrive

        move.l      a0,a1
        move.l      d0,d5

;The meat of the read routine...

.ReadLoop:
        move.l      d1,d3
        divu.w      #(512*11),d3            ;d3.w is track #
        move.l      d3,d4
        clr.w       d4
        swap        d4      ;d4 is byte offset into track

        bsr.b       Seek
        bsr         ReadTrackAndDecode
        tst.l       d7
        bne.s       .End
        move.l      DISK_DecodedBuffer(a5),a0
        add.w       d4,a0
        sub.w       #(11*512),d4
        neg.w       d4
        add.l       d4,d1

;D4.L - number of bytes that could be transferred from this track
;D5.L - number of bytes left in the entire Read request

        cmp.l       d5,d4
        bcc.s       .FinishUp   ;>=
        sub.l       d4,d5
        move.l      d4,d0
        bsr         CopyMem
        add.l       d0,a1
        bra.s       .ReadLoop
```

```
.FinishUp:
        move.l      d5,d0
        bsr         CopyMem     ;use number of bytes left in entire
request
.End:
        movem.l     (sp)+,d0-d5/a0/a1/a4-a6
        rts
.Error:
        moveq       #DISK_BadParameter,d7
        bra.s       .End

SelectDrive:
;Drive number in D2 (0-3)
;Exits in 500ms if disk ready doesn't get set

        movem.l     d0/d2,-(sp)
        addq.w      #3,d2
        or.b        #$78,(a4)   ;Deselect all drives
        bclr        #7,(a4)     ;motor on
        bclr        d2,(a4)     ;select drive X
        move.w      #7875,d0    ;500ms
.L1:    move.b      vhposr(a6),d2
.L2:    btst        #5,ciaapra(a4)          ;check disk ready
        beq.s       .Ready
        cmp.b       vhposr(a6),d2
        beq.s       .L2
        dbra        d0,.L1
.Ready:
        movem.l     (sp)+,d0/d2
        rts

Seek:
;Enter with destination track in d3.b
;Enter with drive number in D2.w

;Note: This routine checks for valid track numbers - if either the
;current track number or the destination is invalid, PANIC.

        movem.l     d0/d3,-(sp)
        cmp.b       #160,d3
        bcc.s       .Error      ;>=
;Set head
        bclr        #2,(a4)     ;set to upper
        btst.l      #0,d3
        bne.s       .skip
        bset        #2,(a4)     ;set to lower
.skip:
        move.b      DISK_CurrentTrack(a5,d2.w),d0
        lsr.b       #1,d0
        cmp.b       #80,d0
        bcc.s       .Error      ;>=
        move.b      d3,DISK_CurrentTrack(a5,d2.w)
        lsr.b       #1,d3
        bset        #1,(a4)     ;set to "out" (lower tracks)
        sub.b       d3,d0
        beq.s       .SideSelectOnly
        bhi.s       .StepIn     ;Go if DISK_CurrentTrack > Destination
        bclr        #1,(a4)     ;set to "in" (higher tracks)
        neg.b       d0
.StepIn:
        bset        #0,(a4)
        bclr        #0,(a4)     ;step head
        bset        #0,(a4)
        bsr.b       delay3
        subq.b      #1,d0
        bne.s       .StepIn
        bsr.b       delay15
.End:   movem.l     (sp)+,d0/d3
        rts
.SideSelectOnly:
        bsr.b       delay1
        bra.s       .End
.Error:
        move.w      #$f00,$dff180
        bra.s       .Error

delay1:
        move.l      d0,-(sp)
        moveq       #15,d0
        bra.s       delay
```

```
delay4:
        move.l    d0,-(sp)
        moveq     #62,d0
        bra.s     delay
delay15:
        move.l    d0,-(sp)
        move.w    #236,d0
        bra.s     delay
delay3:
        move.l    d0,-(sp)
        moveq     #47,d0
delay:
;Enter with magic delay number (minus 1) in d0.w

        push      d1
.L1:    move.b    vhposr(a6),d1
.L2:    cmp.b     vhposr(a6),d1
        beq.s     .L2
        dbra      d0,.L1
        pop       d1
        pop       d0
        rts

SetRegs:
        move.l    DISK_RawBuffer(a5),dskpt(a6)
        move.w    #$6200,adkcon(a6)      ;clear certain bits
        move.w    #$9500,adkcon(a6)      ;set fast, wordsync, mfmprec
        move.w    #$8010,dmacon(a6)      ;enable disk DMA
        move.w    #$4489,dsksync(a6)     ;set magic sync word
        move.w    #$0000,dsklen(a6)      ;force DMA off
        move.w    #$1002,intreq(a6)      ;clear sync+done
        rts

ScanSync:
;Scan for sync mark
;A2 - pointer to raw data - updated
;Error code in D7

        push      a4
        move.l    DISK_RawBuffer(a5),a4
        add.w     #DISK_RawBufSize,a4

.Sync:  cmp.w     #$4489,(a2)+
        beq.s     .found
        cmp.l     a2,a4
        bhi.s     .Sync
        moveq     #DISK_NoSync,d7
        pop       a4
        rts
.found:
        cmp.w     #$4489,(a2)
        bne.s     .ok
        addq.l    #2,a2
        bra.s     .found
.ok:
        pop       a4
        rts

RawRead:
;Perform a raw track read
;Error code returned in D7, if an error occured

        bsr.b     SetRegs
        move.w    #$9CBE,dsklen(a6)
        move.w    #$9CBE,dsklen(a6)      ;read approx. 14,716 bytes

;Fall through to DISK_Wait

DISK_Wait:
        movem.l   d0/d2,-(sp)
        move.w    #7875,d0    ;500ms
.L1:    move.b    vhposr(a6),d2
.L2:    btst      #1,intreqr+1(a6)       ;check disk block done
        bne.s     .Ready
        cmp.b     vhposr(a6),d2
        beq.s     .L2
        dbra      d0,.L1
.Ready:
        moveq     #0,d2
        tst.w     d0
        bpl.s     .NoError
```

```
        moveq     #DISK_NoSync,d7
        move.w    d2,dsklen(a6)
.NoError:
        move.w    d2,dsklen(a6)          ;zero-length DMA transfer
        move.w    #$1002,intreq(a6)      ;clear sync+done
        movem.l   (sp)+,d0/d2
        rts

ReadTrackAndDecode:
;Returns error code in D7

        movem.l   d0-d6/a0-a3,-(sp)

        bsr.b     RawRead
        tst.l     d7
        bne.s     .End

;Decode track
        move.l    DISK_RawBuffer(a5),a2
        move.l    DISK_DecodedBuffer(a5),a3

        moveq     #10,d3     ;# of sectors - 1
        move.l    #$55555555,d2

.SecLoop:
        bsr       ScanSync
        tst.l     d7
        bne.s     .End
        bsr.b     DecodeLong ;get header
        move.l    d0,d1
        subq.l    #8,a2
        bsr       ChecksumHeader        ;generate header checksum
        bsr.b     DecodeLong ;header checksum
        cmp.l     d0,d5
        bne.s     .HeaderError

        and.w     #$ff00,d1  ;mask off sector number
        cmp.w     #$0a00,d1
        bhi.s     .HeaderError
        bsr.b     DecodeLong ;data area checksum
        add.w     d1,d1      ;convert to offset

        lea       (a3,d1.w),a1           ;compute destination

;Decode and checksum data block
        moveq     #127,d6
        moveq     #0,d5
.L1:    move.l    512(a2),d4
        eor.l     d4,d5
        and.l     d2,d4
        move.l    (a2)+,d1
        eor.l     d1,d5
        and.l     d2,d1
        add.l     d1,d1
        or.l      d1,d4
        move.l    d4,(a1)+
        dbra      d6,.L1
        and.l     d2,d5
        add.w     #512,a2

        cmp.l     d0,d5      ;check data area checksum
        bne.s     .DataError
        dbra      d3,.SecLoop
.End:
        movem.l   (sp)+,d0-d6/a0-a3
        rts
.HeaderError:
        moveq     #DISK_BadHeader,d7
        bra.s     .End
.DataError:
        moveq     #DISK_BadData,d7
        bra.s     .End

DecodeLong:
;A2 - ptr to buffer - updated
;D2 - $55555555
;D0 - result

        move.l    d1,-(sp)
        move.l    (a2)+,d0
        move.l    (a2)+,d1
```

```
        and.l   d2,d0
        and.l   d2,d1
        add.l   d0,d0    ;was lsl.l #1,d0
        or.l    d1,d0
        move.l  (sp)+,d1
        rts

EncodeLong:
;Enter with data to be encoded in D0.L
;and pointer to destination in A0 - updated
;Exit with checksum in D5

        movem.l d0-d4,-(sp)
        moveq   #0,d5
        move.l  #$55555555,d4
        move.l  d0,d3
        lsr.l   #1,d0
        bsr.b   Encode
        move.l  d3,d0
        bsr.b   Encode
        and.l   #$55555555,d5
        movem.l (sp)+,d0-d4
        rts

EncodeBlock:
;Enter with pointer to source data in A2 - updated
;Enter with pointer to destination in A0 - updated

;Exit with checksum in D5

        movem.l d0-d4,-(sp)
        moveq   #0,d5
;Encode odd bits
        push    a2
        move.w  #(512/4)-1,d3
        move.l  #$55555555,d4
.L1:    move.l  (a2)+,d0
        lsr.l   #1,d0
        bsr.b   Encode
        dbra    d3,.L1

;Encode even bits
        pop     a2
        move.w  #(512/4)-1,d3
.L2:    move.l  (a2)+,d0
        bsr.b   Encode
        dbra    d3,.L2
        and.l   #$55555555,d5
        movem.l (sp)+,d0-d4
        rts

Encode:
;Enter with longword to code in D0.L and #$55555555 in D4
;uses d0,d1,d2,a0 - not saved

;Accumulates checksum in D5

        and.l   d4,d0
        move.l  d0,d2
        eor.l   d4,d2
        move.l  d2,d1
        add.l   d2,d2
        lsr.l   #1,d1
        bset    #31,d1
        and.l   d2,d1
        or.l    d1,d0
        btst    #0,-1(a0)
        beq.s   .ok
        bclr    #31,d0
.ok:
        eor.l   d0,d5
        move.l  d0,(a0)+
        rts

;This routine corrects the high bit of the current byte based on the
;low bit of the previous byte.

Correct:
        move.b  (a0),d0
        btst    #0,-1(a0)
        bne.s   .ResetClock


        btst    #6,d0
        bne.s   .end
        bset    #7,d0
        bra.s   .end1
.ResetClock:
        bclr    #7,d0
.end1:  move.b  d0,(a0)
.end:   rts

EncodeAndWriteTrack:
;Enter with pointer to source data in A2
;Enter with drive number in d2.w

        movem.l d0-d6/a0/a2,-(sp)
        move.l  DISK_RawBuffer(a5),a0

;Gap = 1660 bytes - 2 bytes for hardware bug
        move.l  #$aaaaaaaa,d1    ;10101010...
        move.w  #414,d0
..      move.l  d1,(a0)+
        dbra    d0,..
        subq.l  #2,a0    ;leave room for 2 extra bytes at the very
end

        moveq   #11,d1    ;number of sectors
        moveq   #0,d3    ;sector count

.SecLoop:
        move.l  #$aaaaaaaa,(a0)
        bsr.b   Correct
        addq.l  #4,a0
        move.l  #$44894489,(a0)+
        move.l  #$ff000000,d0
        moveq   #0,d6
        move.b  DISK_CurrentTrack(a5,d2.w),d6
        swap    d6
        or.l    d6,d0
        move.l  d3,d6
        lsl.l   #8,d6
        or.l    d6,d0
        or.l    d1,d0
        bsr     EncodeLong ;header
        move.l  d5,d4
        moveq   #0,d0
        moveq   #3,d6
..      bsr     EncodeLong ;"OS recovery info"
        dbra    d6,..
        move.l  d4,d0
        bsr     EncodeLong ;header checksum
        move.l  a0,a3    ;save raw data pointer
        addq.l  #8,a0
        bsr     EncodeBlock            ;encode data block
        move.l  d5,d0
        exg     a0,a3
        bsr     EncodeLong ;data block checksum
        bsr     Correct
        move.l  a3,a0
        addq.l  #1,d3
        subq.l  #1,d1
        bne.s   .SecLoop

        move.w  #$aaaa,(a0)
        bsr     Correct    ;extra word to avoid hardware bug

;Physically write the data
        bsr     SetRegs
        cmp.b   #80,DISK_CurrentTrack(a5,d2.w)
        bcs.s   .NoPreComp ;<
        move.w  #$A000,adkcon(a6)    ;140ns precomp
.NoPreComp:
        move.w  #$0400,adkcon(a6)    ;turn OFF wordsync!!!
        move.w  #$DA9E,dsklen(a6)
        move.w  #$DA9E,dsklen(a6)    ;write approx. 13,628 bytes
        bsr     DISK_Wait
        bsr     delay3   ;post-write delay

        IFGT    VERIFYON
        bsr     RawRead
        bsr     Verify
        ENDIF
```

```
        movem.l    (sp)+,d0-d6/a0/a2
        rts

Write:

;Error code returned in D7, as always.

        movem.l    d0-d5/a0/a1/a2/a4-a6,-(sp)
        moveq      #0,d7
        lea        DISK_DataArea(pc),a5
        move.l     #_custom,a6
        move.l     #ciabprb,a4

        move.l     d1,d3
        add.l      d0,d3
        cmp.l      #901120,d3
        bhi.s      .ParmError ;go if >

        bsr        SelectDrive
        btst       #3,ciaapra(a4)        ;check write protect status
        beq.s      .ProtError

        move.l     d0,d5

;The meat of the write routine...

.WriteLoop:
        move.l     d1,d3
        divu.w     #(512*11),d3          ;d3.w is track #
        move.l     d3,d4
        clr.w      d4
        swap       d4          ;d4.l is byte offset into track

        bsr        Seek
        tst.w      d4
        bne.s      .ReadTrack
        cmp.l      #(11*512),d5
        bcc.s      .DontReadTrack        ;>=
.ReadTrack:
        bsr        ReadTrackAndDecode
        tst.l      d7
        bne.s      .End
.DontReadTrack:
        move.l     DISK_DecodedBuffer(a5),a1
        add.w      d4,a1
        sub.w      #(11*512),d4
        neg.w      d4
        add.l      d4,d1

;D4.L - number of bytes that could be transferred from this track
;D5.L - number of bytes left in the entire Read request

        cmp.l      d5,d4
        bcc.s      .FinalWrite           ;>=
        sub.l      d4,d5
        move.l     d4,d0
        bsr        CopyMem
        add.l      d0,a0
        move.l     DISK_DecodedBuffer(a5),a2
        bsr        EncodeAndWriteTrack
        tst.l      d7
        bne.s      .End
        bra.s      .WriteLoop

.FinalWrite:
        move.l     d5,d0
        bsr        CopyMem
        move.l     DISK_DecodedBuffer(a5),a2
        bsr        EncodeAndWriteTrack
.End:
        movem.l    (sp)+,d0-d5/a0/a1/a2/a4-a6
        rts
.ParmError:
        moveq      #DISK_BadParameter,d7
        bra.s      .End
.ProtError:
        moveq      #DISK_WriteProtected,d7
        bra.s      .End
```

```
MotorOff:

;MotorOff turns off all drive motors and leaves all drives deselected

        push       a4
        move.l     #ciabprb,a4
        bset       #7,(a4)    ;motor off
        or.b       #$78,(a4)  ;deselect all drives
        and.b      #$87,(a4)  ;select all drives
        or.b       #$78,(a4)  ;deselect all drives
        pop        a4
        rts

Inquire:
;Find out what drives are in the system
;Exit with "drive map" in D0 (bit 0 = drive 0, bit 1 = drive 1, etc.)

;See the Hardware Reference Manual, Appendix E for a description of
;the procedure.

;Note: Drive zero is always present. In fact, the identification
;scheme does not work with drive zero.

        movem.l    d1-d5/a0,-(sp)
        moveq      #4,d3
        moveq      #1,d2      ;drive count
        moveq      #1,d0      ;drive map (drive zero always present)
        moveq      #7,d1
        move.l     #ciabprb,a0

        or.b       #$78,(a0)  ;deselect all drives

.InquireLoop:
        moveq      #31,d4
        moveq      #0,d5      ;identification longword

        bclr       d1,(a0)    ;motor on
        bset       d3,(a0)    ;deselect drive
        bclr       d3,(a0)    ;select drive

        bset       d1,(a0)    ;motor off
        bset       d3,(a0)    ;deselect drive
        bclr       d3,(a0)    ;select drive
        bset       d3,(a0)    ;deselect drive

.ReadIdent:
        bclr       d3,(a0)    ;select drive
        btst       #5,ciaapra(a0)        ;test ready
        beq.s      .zero
        bset       d4,d5
.zero:  bset       d3,(a0)    ;deselect drive
        dbra       d4,.ReadIdent

        tst.l      d5
        bne.s      .Skip
        bset       d2,d0
.Skip:  addq.l     #1,d3
        addq.l     #1,d2
        cmp.b      #4,d2
        bne.s      .InquireLoop
        movem.l    (sp)+,d1-d5/a0
        rts

SeekZero:
;Place all drives in the system on track zero
;The previous track position of each drive is recorded in DISK_OldTrack
;(4 bytes, one for each drive)

;Also sets CIA data direction registers - very important in some cases!

        movem.l    d0-d2/a0/a4-a6,-(sp)
        lea        DISK_DataArea(pc),a5
        lea        DISK_OldTrack(a5),a0
        move.l     #ciabprb,a4
        move.l     #_custom,a6

;set up CIA data direction registers
        move.b     #$03,ciaaddra(a4)
        move.b     #$FF,ciabddrb(a4)

        bsr.b      Inquire
```

```
                lsl.l       #3,d0
                moveq       #3,d1
.SeekLoop:
                btst        d1,d0
                beq.s       .NoDrive

                bclr        d1,(a4)     ;select
                bset        #1,(a4)     ;set to "out" (lower tracks)
                moveq       #0,d2       ;initialize old track counter
.StepLoop:
                btst        #4,ciaapra(a4)      ;check track zero flag
                beq.s       .EndStepLoop
                addq.l      #1,d2
                bset        #0,(a4)
                bclr        #0,(a4)     ;step head
                bset        #0,(a4)
                bsr         delay4
                bra.s       .StepLoop

.EndStepLoop:
                add.l       d2,d2
                move.b      d2,(a0)
                bset        d1,(a4)     ;deselect
.NoDrive:
                addq.l      #1,a0
                addq.l      #1,d1
                cmp.b       #7,d1
                bne.s       .SeekLoop
                clr.l       (a5)
                bsr         delay15
                movem.l     (sp)+,d0-d2/a0/a4-a6
                rts

                IFGT        VERIFYON
Verify:
;Verify the integrity of a raw track
;Returns error code in D7, if there is an error

                movem.l     d0/d2-d5/a2,-(sp)
                move.l      DISK_RawBuffer(a5),a2
                moveq       #10,d3      ;# of sectors - 1
                move.l      #$55555555,d2
.VerLoop:
                bsr         ScanSync
                tst.l       d7
                bne.s       .VerError
                bsr.b       ChecksumHeader      ;generate header checksum
                move.l      d5,d4
                bsr         DecodeLong  ;header checksum
                cmp.l       d0,d4
                bne.s       .VerError   ;header error
                bsr         DecodeLong  ;data area checksum
                move.l      d0,d4
                move.w      #1024/4,d0
                bsr.b       ChecksumWords       ;generate data area checksum
                cmp.l       d5,d4
                bne.s       .VerError   ;data error
                dbra        d3,.VerLoop
.End:
                movem.l     (sp)+,d0/d2-d5/a2
                rts
.VerError:
                moveq       #DISK_VerifyError,d7
                bra.s       .End
                ENDIF

ChecksumHeader:
                moveq       #10,d0
ChecksumWords:
;D0 - number of longwords to checksum
;A2 - pointer to raw data - updated
;D2 - $55555555
;Exit with checksum in D5

                movem.l     d0-d1,-(sp)
                moveq       #0,d5
                move.l      (a2)+,d1
                eor.l       d1,d5
                subq.w      #1,d0
                bne.s       ..
```

```
                and.l       d2,d5
                movem.l     (sp)+,d0-d1
                rts

CopyMem:

;A0 - source
;A1 - destination
;D0 - size

                movem.l     d0/d1/a0/a1,-(sp)
                moveq       #12,d1
                cmp.l       d1,d0
                bcs.s       .L6
                move.l      a0,d1
                btst        #0,d1
                beq.s       .L1
                move.b      (a0)+,(a1)+
                subq.l      #1,d0
.L1:            move.l      a1,d1
                btst        #0,d1
                bne.s       .L6
                move.l      d0,d1
                and.w       #3,d1
                move.w      d1,-(sp)
                moveq       #$60,d1
                cmp.l       d1,d0
                bcs.s       .L3
                movem.l     d1-d7/a2-a6,-(sp)
.L2:            movem.l     (a0)+,d1-d7/a2-a6
                movem.l     d1-d7/a2-a6,(a1)
                moveq       #$30,d1
                add.l       d1,a1
                sub.l       d1,d0
                cmp.l       d1,d0
                bcc.s       .L2
                movem.l     (sp)+,d1-d7/a2-a6
.L3:            lsr.l       #2,d0
                beq.s       .L5
                subq.l      #1,d0
                move.l      d0,d1
                swap        d0
.L4:            move.l      (a0)+,(a1)+
                dbra        d1,.L4
                dbra        d0,.L4
.L5:            move.w      (sp)+,d1
                beq.s       .L9
                moveq       #0,d0
                bra.s       .L8
.L6:            move.w      d0,d1
                swap        d0
                bra.s       .L8
.L7:            move.b      (a0)+,(a1)+
.L8:            dbra        d1,.L7
                dbra        d0,.L7
.L9:            movem.l     (sp)+,d0/d1/a0/a1
                rts

RestoreHeadPosition:

;This routine moves the heads to their previous positions, saved by
;a call to SeekZero. It also does a MotorOff.
;Note that this routine depends on the DISK_OldTrack field to be
;initialized to $FFs to avoid messing with drives that don't exist.
;(SeekZero does the work of calling Inquire).

                movem.l     d0-d3/a0/a4-a6,-(sp)
                move.l      #ciabprb,a4
                move.l      #_custom,a6
                bsr         MotorOff
                moveq       #3,d0       ;number of drives - 1
                moveq       #3,d1       ;bit number in ciabprb
                moveq       #0,d2       ;drive number (0-3)
                lea         DISK_DataArea(pc),a5
                lea         DISK_OldTrack(a5),a0

.Seek1:         move.b      (a0)+,d3
                bmi.s       .Seek2
                bclr        d1,(a4)     ;select drive
                bsr         Seek
                bset        d1,(a4)     ;deselect
```

```
.Seek2:    addq.l   #1,d2
           addq.l   #1,d1
           dbra     d0,.Seek1
           movem.l  (sp)+,d0-d3/a0/a4-a6
           rts

;          END
```

## Listing Two    disktest.s

```
;disktest.s - test program for disk.s, low level disk I/O
;Demonstrates taking over with DR_GETUNIT

           exeobj
           objfile  'disktest.exe'

           move.l   4,a6

;Allocate some chip RAM for our track buffer
           moveq    #MEMF_CHIP,d1
           move.l   #DISK_RawBufSize,d0
           SYS      AllocMem
           move.l   d0,rawbufptr
           beq      Abort1

           moveq    #0,d1
           move.l   #(11*512),d0
           SYS      AllocMem
           move.l   d0,cookedbufptr
           beq      Abort2

           lea      diskname(pc),a1
           SYS      OpenResource
           move.l   d0,resourceptr
           lea      myunit(pc),a2
           lea      replyport(pc),a3
           move.l   a3,MN_REPLYPORT(a2)

;Initialize the ReplyPort

           sub.l    a1,a1
           SYS      FindTask
           move.l   d0,MP_SIGTASK(a3)
           moveq    #-1,d0
           SYS      AllocSignal
           move.b   d0,MP_SIGBIT(a3)
           move.b   #NT_MSGPORT,LN_TYPE(a3)
           lea      MP_MSGLIST(a3),a0
           NEWLIST  a0

.GetUnit:
           move.l   a2,a1
           move.l   resourceptr(pc),a6
           jsr      DR_GETUNIT(a6)
           tst.l    d0
           bne.b    .ExitGetUnit
           move.l   a3,a0
           move.l   4,a6
           SYS      WaitPort
           bra.b    .GetUnit
.ExitGetUnit:

;Ok, we now have complete control over the drive

           bsr      SeekZero  ;Step the heads to track zero

;Set up buffer pointers for RWTS code

           lea      DISK_DataArea(pc),a2
           move.l   rawbufptr(pc),DISK_RawBuffer(a2)
           move.l   cookedbufptr(pc),DISK_DecodedBuffer(a2)

;--------- Test code starts here ---------
```

```
;Copy a disk from drive 0 to drive 1

;WARNING: Remove one of the disks after the copy or AmigaDos will
;crash! (due to both disks having the same exact name and creation time)

           move.l   #(0),d1   ;offset
           moveq    #79,d3

.CopyLoop:
           move.l   #512*22,d0  ;length
           move.l   #0,d2       ;drive
           move.l   #MyBuffer,a0          ;buffer
           bsr      Read
           tst.l    d7
           bne      .Error

           move.l   #512*22,d0  ;length
           move.l   #1,d2       ;drive
           move.l   #MyBuffer,a0          ;buffer
           bsr      Write
           tst.l    d7
           bne      .Error
           add.l    #(512*22*1),d1
           dbra     d3,.CopyLoop
.Error:

;--------- test code ends ---------

           bsr      RestoreHeadPosition  ;quite important for 'clicking'
                                         ;drives

           move.l   resourceptr(pc),a6
           jsr      DR_GIVEUNIT(a6)
           move.l   4,a6

;Normal exit

           lea      replyport(pc),a0
           moveq    #0,d0
           move.b   MP_SIGBIT(a0),d0
           SYS      FreeSignal

           move.l   #(11*512),d0
           move.l   cookedbufptr(pc),a1
           SYS      FreeMem

Abort2:
           move.l   #DISK_RawBufSize,d0
           move.l   rawbufptr(pc),a1
           SYS      FreeMem

Abort1:
           moveq    #0,d0
           rts

;************************************************************
;************************* Data area ************************
;************************************************************

diskname:  dc.b     'disk.resource',0
           even
resourceptr:        dc.l     0
rawbufptr: dc.l     0
cookedbufptr:       dc.l     0
myunit:             dcb.b    MN_SIZE,0
replyport: dcb.b    MP_SIZE,0
           even

           include  "disk.s"

           section  mybuffer,bss

MyBuffer:  ds.b     512*22

           end
```
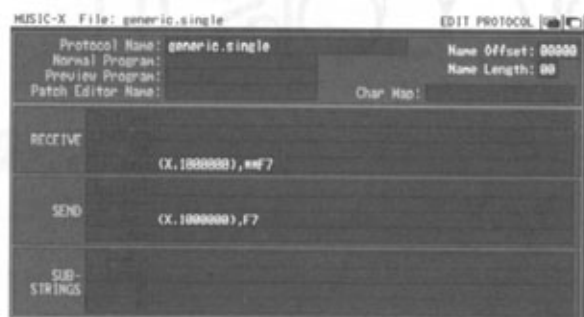
```
MUSIC-X  File: generic.single                    EDIT PROTOCOL ▣▢

    Protocol Name: generic.single           Name Offset: 00000
      Normal Program:                        Name Length: 00
     Preview Program:
   Patch Editor Name:              Char Map:


  RECEIVE
                  (X.1000000),==F7


  SEND
                  (X.1000000),F7


  SUB-
  STRINGS
```

The important part of this protocol is found in the Data strings in the SEND and RECEIVE panels. Briefly, the RECEIVE Data entry says to read data in 16-megabyte ($1000000-byte) chunks until an $F7 is received. The SEND Data entry says to send 16-megabyte chunks of data until there is none left, and then send $F7. (Note: $F7 is a special MIDI byte meaning "end of system exclusive data.")

In other words, this protocol is rather mindless. It grabs or sends as much data as it can find, regardless of what the data looks like. If you could hand it a DeluxePaint file, for example, this protocol would happily send it to your MIDI synthesizer. (No, the picture would not appear on your synth's front panel!)

### What Does a Smart Protocol Need to Know?

To make your protocol smarter, you need to tell it some vital information, such as:

- How long (in bytes) is the data for 1 patch?
- How do I ask for a patch?
- How do I recognize the beginning of a legal patch?
- Which byte represents the patch number?
- Where is the patch name located, and how long is it?
- When I am sent a patch, do I need to respond?
- When I send a patch, should I expect an acknowledgement?
- If the user aborts a transfer, should I inform the instrument?

You do not need to answer all of the questions: only those that apply to your instrument. For example, some instruments transmit all their data in one big message, but others require an elaborate "conversation" between the sender and the recipient. You must read your instrument's system-exclusive documentation to determine which questions need to be answered. For some help with this, see Phil Saunders' "Medley" columns in the April and May 1991 issues of *Amazing Computing*.

### Building a Protocol: One Approach

After writing several protocols, I discovered that there is a pattern to the process. Here's a brief outline.

1. Obtain a copy of your instrument's system exclusive documentation. Specifically, you'll need to know the proper bytes to send when requesting a patch from the instrument, and the format of a patch dump itself. If this information is not included in the owner's manual, then contact the manufacturer. Many manufacturers will send you the information free of charge.

2. Use Generic.Single to receive a patch from your instrument. You will need to initiate the patch transfer yourself by pressing the appropriate buttons on your instrument.

3. If the manufacturer's documentation does not tell you the size of a patch dump, then look at a patch's data using the Librarian's Edit Entry command, and determine the size yourself. You may also wish to examine the patch dump in more detail.

4. Starting with a copy of Generic.Single, write the RECEIVE part of the protocol. Test it.

5. Write the SEND part of the protocol.

6. Make Music-X extract and display the names of patches as the are received, if applicable.

The rest of this article will concentrate on items (4) through (6). Before we can construct the SEND and RECEIVE parts of the protocol, however, we must learn about Music-X variables. There are three kinds: numeric, data, and string.

### Numeric Variables

A numeric variable is much like one found in programming languages or high-school algebra. It is a single letter (case-insensitive) that stands for a numeric value. Some of these variables may have their values set by you on the Librarian Page:

P   The patch number that you set on the Librarian Page.
N   The MIDI channel that you set on the Librarian Page

and others have their values maintained by Music-X:

M   Total blocks sent.
V   The last value received.
K   Checksum of a specified sequence of bytes.

A complete list of numeric variables is found in the manual.

### Data Variables

The two data variables, X and Y, store sequences of data sent or received by the Librarian. X stores data in its raw, unmodified form. Y stores data in "nibblized" form, discussed later. The manual discusses the formal syntax, but here are a few examples to get us warmed up.

In general, to store k bytes of data, you use the expression:

(X.k)

For example, this is how you tell Music-X to send or receive 25 ($19) bytes of data:

(X.19)

# Writing Protocols in Music-X

*by Daniel J. Barrett*

Congratulations! After working hard all summer playing keyboards with your band, you finally have saved up enough money to buy that fancy WhizzySynth 3000 sitting in the music store. You bring it home and happily start making your own patches. At the end of the day, you decide to store those patches on an Amiga disk using your trusty Music-X Librarian. But after looking through your "Protocols" directory, you find that the Music-X Librarian doesn't support the WhizzySynth 3000 for patch transfers! Oh no... what can you do now?

Well, it's time to write your own protocol. Music-X's Librarian is billed as "universal"; it can communicate with any MIDI instrument, provided that you supply the right information using the built-in Protocol Editor. Although this process is documented in the Music-X manual, it is still tricky to do. In addition, the manual is more of a reference than a tutorial.

This article will teach you how to write a Music-X protocol and contains plenty of examples. I won't repeat straightforward information found in the manual, so I recommend having it near you while you work through this article. Also, I won't cover every single detail about protocols; instead, I'll concentrate on the more practical uses (and a few less common ones) so you can start writing protocols as quickly as possible.

Since this is a technical journal, I'll assume that you know how to use an ASCII table and hexadecimal numbers. In the text, all hex numbers will be preceded by a dollar sign to distinguish them from decimal numbers. (For example, hex $42 equals 66 in base ten.) In addition, some familiarity with MIDI system exclusive data will be helpful.

## What is a Protocol?

First of all, what is a protocol? It is a general description of what your instrument's patch data looks like. *(When I say "instrument" here, I really mean any MIDI device that is capable of sending and receiving system exclusive data.)* Equipped with this information, Music-X can send and receive your patch data. If the incoming MIDI bytes don't match the given description, then Music-X will inform you that an error has occurred.

Both libraries and protocols may be saved as separate files on your disk. The following diagram illustrates the relationship of library and protocol files *(see figure one).*



**Figure One**  Relationship of library & protocol files
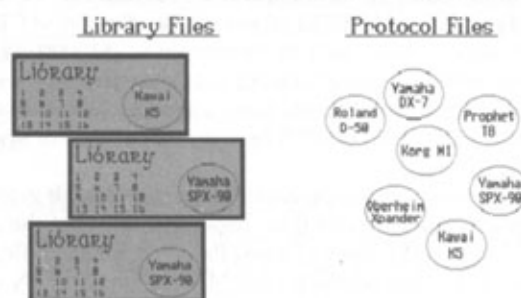
Although library files are physically separate from protocol files, every library file has a COPY of some protocol inside it. If you modify the copy inside a library, and then save the library, the protocol file on disk is not affected. Similarly, modifying the protocol file on disk does not affect any libraries. Thus, if you have a protocol that is used by ten different libraries, and you want to modify that protocol, you must change it in all 10 libraries! This is a shortcoming of the implementation.

Music-X comes supplied with protocols for several popular instruments. In particular, there is a "generic" protocol called Generic.Single that works for almost all instruments. "Well," you say, "if this generic protocol is so versatile, why not use it for all my instruments?" One reason is that Generic.Single cannot request patches from your instrument—you must initiate the transfer from the instrument's front panel (assuming this is possible). A second reason is that it cannot distinguish legal from illegal data. This means that if there is a transmission error between your instrument and your Amiga, the generic protocol will not notice.

However, Generic.Single is a good starting point for writing your own specific protocol. Let's take a look at the generic protocol by making a generic library. Run Music-X, and choose Librarian from the Mode menu. Then choose New... from the File menu, move to the appropriate directory, and select the file Generic.Single. Finally, choose Modify Protocol from the Edit menu, and we have arrived at the Protocol Editor *(see figure two):*

Here is how you tell Music-X to SKIP past 25 bytes of data, not storing it:

(X.19.19)

Here is how to tell Music-X to break 50 bytes of data into nybbles, delete all the most-signficant nybbles, and concatenate the remaining data into 25 ($19) bytes:

(Y.19)

### String Variables

String variables can hold any text. The most commonly used string variable is Prefix, whose value is notated as PR in your data. Typically, system exclusive messages for the same instrument will begin with the same few bytes. To save typing (and make modification easier later), store those bytes inside the PR variable. Our later examples will all do this.

A second, very useful string variable is the Program string, which is represented as PG. To use it, fill in the strings labeled Normal Program and Preview Program with anything you like. The value of the PG variable will be either of these two strings. You toggle between the two values by pressing the PREVIEW button on the Librarian Page. This provides a convenient way for the user to change the behavior of the protocol without entering the Protocol Editor.

Two general-purpose string variables are known merely as #1 and #2. Fill them with any data you like, and use them anywhere on the SEND and RECEIVE panels. If you don't actually need these strings for protocol data, they provide a convenient place to write comments to yourself about the protocol.

### Writing the Send and Receive Strings

Filling in the SEND and RECEIVE panels is perhaps the most difficult part of protocol writing. You must describe a MIDI "conversation" that allows your instrument and Music-X to communicate. Music-X breaks down the communication into five types of messages: Initiate, Confirm, Ack, Data, and Cancel. Some instruments use only a subset of these messages, but others require an ongoing conversation (known as "handshaking") while data is being transferred (see figure three).

An Initiate string says, "Hello, I would like to begin a patch transfer." To determine its value, consult your instrument's system exclusive documentation, and look for a message called

"Request to send," "Request for bulk dump," or something similar. For the SEND Initiate string, one often uses the header from a patch dump.

If your instrument does not require any handshaking, then you will need to use only the Initiate and Data strings, and you are now ready to write your Data strings. These contain the X and/or Y data variables for sending or receiving the patch data and storing it as a library entry. The case studies in the next section will provide several examples.

If you instrument does require handshaking, then you must also provide Confirm and/or Ack ("Acknowledge") strings. These strings each represent the message "I am ready!" Confirm is sent by the instrument (and so Music-X must be told how to recognize it), and Ack is sent by Music-X. Depending on the system exclusive implementation, your instrument expects a particular sequence of Confirm/Data/Ack messages. The Music-X manual is really quite good about explaining this mechanism; however, you will need to spend some time with your instrument's system exclusive documentation to figure out what messages need to be used.

Finally, if your instrument requires a special message if the patch transfer is aborted, you should fill a value for the Cancel string.

### CASE STUDY: Sequential Circuits Prophet T8 synthesizer

Sequential Circuits was the company that essentially invented MIDI. The T8 has a very simple MIDI implementation, and the patches have no names, so the protocol itself is quite short.

Reading the T8 manual, we find that the following message will ask the T8 to send a particular patch via MIDI:

| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $01 | Sequential Circuits' manufacturer ID. |
| $00 | Request to send a patch, please. |
| ... | The patch number. |
| $F7 | "End of system exclusive" byte. |

Music-X needs to send this message when it wants to receive a patch. So, we fill in the following data in the RECEIVE Initiate box:

F0, 01, 00, P, F7

Now, Music-X must be prepared to receive data from the T8. A patch contains 68 ($44) bytes of data, followed by $F7. So, in the RECEIVE Data box, we write:

(X.44), F7

Now, we could have written (X.45) and grabbed the $F7 byte at the same time. However, by writing $F7 explicitly, we cause Music-X to check that an $F7 is received as the 69th byte. Also, since we have specified the $F7 separately, not as part of an "X" variable, it will not be stored as part of the patch. We must remember to append an $F7 when we send our patch data back to the T8.

To fill in the SEND panel, we must examine the T8 patch data dump format:
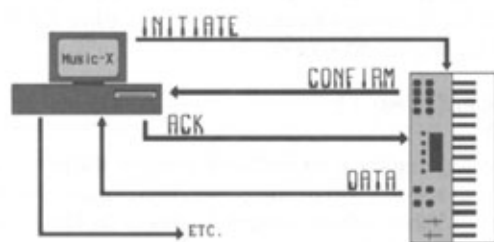
| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $01 | Sequential Circuits' manufacturer ID. |
| $03 | This is data for one patch. |
| ... | The patch number. |
| ... | 68 bytes of patch data. |
| $F7 | "End of system exclusive" byte. |

To transmit the patch back to the instrument, we could simply send everything we have stored. However, this is a bad solution because the original patch number is stored inside the patch data. Thus, we would be able to send the patch back to its original location ONLY! This is not very versatile, especially since the Librarian will let us change the patch number on the Librarian Page if we use the "P" variable.

To solve this problem, we'll use the first part of the data dump format as our SEND Initiate string, skip the original first four bytes of the stored patch, and then send the rest of the data. So our SEND Initiate value is now:

F0, 01, 03, P

and our SEND Data value, which skips past the first 4 bytes, is:

(X.4.4), (X.40), F7

Note that we remembered to append the $F7 that earlier we chose not to store.

Since the first two bytes of both Initiate values are the same—$F0, $01—let's define the PR ("Prefix") string variable to have this value, and use PR in both Initiate commands. They become:

RECEIVE Initiate: PR, 00, P, F7

SEND Initiate: PR, 03, P

and our T8 protocol is now complete (see figure four).

## CASE STUDY: Oberheim Xpander synthesizer

The Xpander's protocol is a bit more complicated than the T8's because Xpander patches contain names that we can extract and display on the Librarian Page. To build this protocol, we consult the "Oberheim Xpander/Matrix-12 MIDI Specification," part number 950038 from Oberheim.
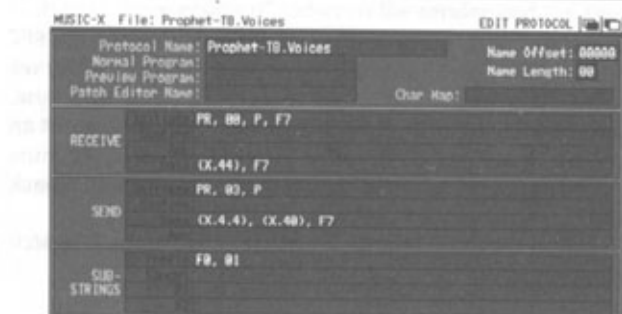
Using Generic.Single to grab 1 patch, and reading the documentation, we find out the following information:

• A single patch dump contains 398 bytes plus $F7.
• The patch name is found in bytes 382-398.
• The format for requesting 1 single patch is:

| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $10 | Oberheim's manufacturer ID. |
| $02 | This is an Xpander. |
| $00 | Send me a single patch, please. |
| ... | The patch number (0-99). |
| $F7 | "End of system exclusive" byte. |

The format of a single patch dump is:

| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $10 | Oberheim's manufacturer ID. |
| $02 | This is an Xpander. |
| $01 | Here comes a patch. |
| $00 | It is a single patch. |
| ... | The patch number (0-99) |
| ... | 392 bytes of patch data. |
| $F7 | "End of system exclusive" byte. |

Now we are ready to start writing the protocol. Since both the SEND and RECEIVE command will begin with the same bytes, we define the PR ("Prefix") variable once again—this time to be:

F0, 10, 02

On the RECEIVE panel, we set the Initiate value to:

PR, 0, 0, P, F7

We now expect an entire patch dump of 398 ($18E) bytes to follow, ending with an $F7, so we can set the RECEIVE Data value to:

(X.18E), F7

For sending data, we use the same trick as in the T8 case study: skip over the first several bytes (containing the old patch number) and substitute our own in the SEND Initiate value. This value is:
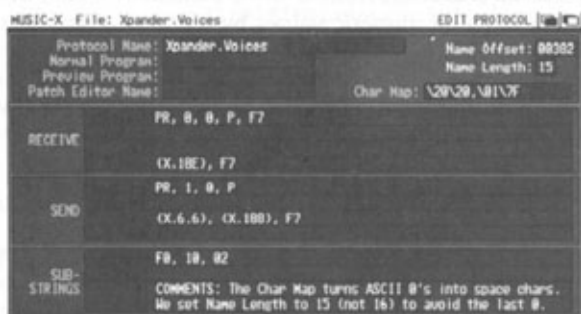
PR, 1, 0, P

We now skip the first 6 bytes of data (the header) and send only the 392 ($188) bytes of patch data itself, plus the $F7 we stripped off during the RECEIVE.

(X.6.6), (X.188), F7

Finally, let's tell the protocol how to locate the patch name. It is found in bytes 382-398 of the patch data. Thus, give Name Offset a value of 382 and Name Length a value of 16. The Librarian will now extract and display the name of each patch as it is received (see figure five).

### Figure Four   T8



MUSIC-X File: Prophet-T8.Voices                          EDIT PROTOCOL

Protocol Name: Prophet-T8.Voices
Normal Program:                                          Name Offset: 00000
Preview Program:                                         Name Length: 00
Patch Editor Name:                    Char Map:

RECEIVE      PR, 00, P, F7
             (X.44), F7

SEND         PR, 03, P
             (X.4.4), (X.40), F7

SUB-         F0, 01
STRINGS

MUSIC-X  File: Xpander.Voices                                    EDIT PROTOCOL ▣▢

| | | |
|---|---|---|
| Protocol Name: Xpander.Voices | | Name Offset: 00382 |
| Normal Program: | | Name Length: 15 |
| Preview Program: | | |
| Patch Editor Name: | | Char Map: \20\20,\01\7F |
| **RECEIVE** | PR, 0, 0, P, F7 | |
| | (X.18E), F7 | |
| **SEND** | PR, 1, 0, P | |
| | (X.6.6), (X.188), F7 | |
| **SUB-STRINGS** | F0, 10, 02 | |
| | COMMENTS: The Char Map turns ASCII 0's into space chars. We set Name Length to 15 (not 16) to avoid the last 0. | |

Normally, we would be done now. However, Xpander patch names are stored in a particular manner that requires us to do a little more work. At the moment, only the first character of the patch name will actually be displayed on the Librarian Page. We shall solve this problem in Example #2 under HANDLING PATCH NAMES, below.

## CASE STUDY: *Yamaha SPX-90*
### *(or SPX-90II) digital effects unit*

*(In order to transmit patch data from an SPX-90, you must first turn its MIDI THRU jack into a MIDI OUT. This is done by removing the top cover (8 screws in all) and flipping switch number SW105 to the "T" position.)*

In this protocol, we will use checksums and see an alternate, more reliable, method for handling the Data strings.

To request a patch from the SPX-90, the command is:

| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $43 | Yamaha's manufacturer ID. |
| $2n | Use MIDI channel n, increased by $20. |
| $7E | Send me some kind of patch dump. |
| "LM 8332" | I am an SPX-90 (there are 2 spaces). |
| "M" | Give me one patch dump. |
| ... | The patch number. |
| $F7 | "End of system exclusive" byte. |

By now, we are experts and can convert this into a RECEIVE Initiate string quickly, except for the line that says "Use MIDI channel n." Music-X provides the N variable which always contains the number of the MIDI channel that we set on the Librarian Page. Once we add $20 to it, we're ready to type in the string:

    F0, 43, (20+n), 7E, "LM 8332", "M", P, F7

We will need some of these bytes again later, so let's store the first two bytes as the Prefix, and the two strings as Substring #1. This makes our RECEIVE Initiate value:

    PR, (20+n), 7E, #1, P, F7

The format of the SPX-90 patch dump is the most complicated we've seen so far:

| | |
|---|---|
| $F0 | "Begin system exclusive" byte. |
| $43 | Yamaha's manufacturer ID. |
| *n* | Use MIDI channel n. |
| $7E | This is some kind of patch dump. |
| $00 | Together with the next byte... |
| $58 | ...this is the data length: 88 bytes. |
| "LM 8332" | I am an SPX-90 (there are 2 spaces). |
| ... | The patch number. |
| ... | 32 bytes holding the 16-character patch name. |
| ... | 46 bytes of patch data. |
| ... | Checksum |
| $F7 | "End of system exclusive" byte. |

Our RECEIVE Data string needs to read 16 ($10) bytes of header, 32 ($20) bytes of patch name, 46 ($2E) bytes of patch data, and 1 checksum byte. Using the same method as the previous two protocols, and ignoring the checksum issues for now, we would have a RECEIVE Data value of:

    (X.10), (X.20), (X.2F), F7

or more simply:

    (X.5F), F7

However, this time we will build more intelligence into the RECEIVE Data string by describing the format of the patch dump. This approach has two advantages: it checks the incoming data more closely, and it saves space by not storing the header bytes inside each patch. A small disadvantage is that the patch library data is now more heavily dependent on the protocol; thus, if you plan to write your own programs to interpret Music-X library files, you may have a tougher time doing it.

There are 32+46+1 ($4F) bytes of data after the patch number. Once again ignoring the checksum, we write the RECEIVE Data string as follows:

    F0, 43, N, 7E, 0, 58, "LM 8332", "M", P, (X.4F), F7

This is now an accurate description of the data we shall expect, so Music-X will complain if it reads any non-matching bytes. Using our PR and #1 substrings, this becomes:

    PR, N, 7E, 0, 58, #1, P, (X.4F), F7

Since everything before the (X.4F) is not captured by the X data variable, the header will not be stored inside each library entry. This fact is the reason for the advantages and disadvantages I listed earlier.

Now, let's handle the checksum byte. Yamaha forms its checksum value by adding all the bytes between #1 and (X.4E) inclusive, negating the value, applying a logical "and" operation with $7F, and truncating to one byte. This operation is notated as (-K&7F.1). If the checksum received does not match this value, Music-X complains.

To form this checksum, we surround the relevant bytes with curly braces. Thus, our finished RECEIVE Data string is:

PR, N, 7E, 0, 58, {#1, P, (X.4E)}, (-K&7F.1), F7

Now it's time to build the SEND portion of the protocol. We must remember that the header bytes and the checksum have not been stored in the library entry, so we must construct and send them manually. Well, guess what? We can use the same patch description we made for the RECEIVE Data string. There is no need for any SEND Initiate string in this case.

To save typing, we store the RECEIVE Data string as string #2, and then just put #2 in our RECEIVE Data and SEND Data strings. We are now done except for handling the patch name, which we discuss in Example #3 of HANDLING PATCH NAMES, below (see figure six).

## Handling Patch Names

If your instrument's patch data contains the name of the patch, Music-X can usually extract the name and display it on the Librarian Page the instant that the patch is received. In order for this to work, you may need to specify some extra information:

1. WHERE the patch name is located in the patch dump;
2. HOW LONG the patch name is;
3. In WHAT FORM the patch name is encoded: the "Character Map."

The first two items are pretty intuitive: after all, Music-X needs to know where the patch name begins and ends. This information is placed into the Name Offset and Name Length gadgets as explained in the manual. (Note: these two numbers must be in decimal, even though every other number in the Protocol Editor is in hexadecimal.) The Character Map, however, can be harder to understand, so I will now explain it in detail and include several examples.

Even though Music-X knows WHERE your patch name is, it may not know how to INTERPRET it. If your MIDI instrument uses ASCII and stores its patch name one character per byte, then no special interpretation will be necessary. However, suppose your instrument uses the values 1 through 26 to represent the letters 'A' through 'Z', values 27 through 36 to represent the digits '0' through '9', and 37 to to be a question mark character. This is NOT the ASCII code. So, you need to tell Music-X how to TRANSLATE between your MIDI instrument's internal code for the patch name, and the standard ASCII code that Music-X (and most computers) use. This is done with a Character Map.

To construct a Character Map, picture the numbers 0 through 127 all lined up in a row. These are the numbers that your instrument uses for characters in patch names. Beneath each number, write the character that the number SHOULD represent to Music-X. The resulting two-row table is a Character Map. Here's the Map for our example above:

```
0 1 2 3 4 5 6 7 8 9 10 ... 25 26 27 28 29 30 31 32 33 34 35 36 37
  A B C D E F G H I J  ...  Y  Z  0  1  2  3  4  5  6  7  8  9  ?
```

Now that we have the table, how do we fill in the Char Map field on the Protocol Editor screen? This field contains a list of RANGES OF CHARACTERS, separated by commas. For ex-

ample, AZ means "all characters from 'A' to 'Z,' inclusive," and 06 means "all characters from '0' to '6,' inclusive." You may also indicate a range of numeric values by using two hexadecimal numbers preceded by backslashes.

Now look at your two-row table and pack your second row into ranges, as described above. In our example given, you wind up with this Character Map:

\00\00,AZ,09,??

The first range, \00\00, is only one character long. It says that the value 0 should be translated into 0. Is this detail necessary? Yes, because a Character Map is always assumed to begin at zero. If we used the Map

AZ,09,??

instead, then the value 0 would be translated into 'A,' 1 into 'B,' etc. This is wrong —it's off by 1.

The second range, AZ, says that the values 1-26 get translated into the characters 'A' to 'Z.' The third range, 09, translates the values 27-36 into the characters '0' through '9.' The last range, ??, is only one character long, and translates the value 37 into a question mark character. What happens to the values 38-127? The manual does not say, so I play it safe and specify that they translate to themselves:

\00\00,AZ,09,??,\26\7F

Character Maps are versatile but have a few serious limitations. First of all, they cannot handle characters that are not stored one per byte. For example, the Prophet VS synthesizer stores its name in tightly packed, 5-bit characters, which Music-X cannot translate. Second, Character Maps are insufficient if your instrument's internal character set is wildly unlike ASCII. For instance, if a synth has an internal character set like this:

```
0   1   2   3   4   5   6   7 ...
'A' 'Q' '7' '8' 'R' 'M' '2' '*' ...
```

with characters placed arbitrarily in the table, there is no Character Map that can represent this translation.

But, you cry, why can't we define a separate range for each character, containing ONLY that character, like this?

AA,QQ,77,%%,RR,MM,22,**, ...

In theory, you can; unfortunately, the Char Map gadget can hold only 79 character! Thankfully, today's MIDI instruments are constructed around popular microprocessors that use ASCII. Perhaps a future version of Music-X will address these limitations.

Here are some examples of constructing character maps.

## Example 1: Not Too Tricky

Suppose your MIDI instrument uses values 28-36 for the digits '1' through '9', 37 for '0', 53-78 for small letters a-z, and 83-108 for capital letters A-Z.

\00\1B,19,00,\26\34,az,\4F\52,AZ,\6D\7F

The ranges are 0-27 ($0-$1B) untouched, the digits '1' to '9,' the digit '0' in its own range, 38-52 ($26-$34) untouched, the letters a-z, 79-82 ($4F-$52) untouched, and the letters A-Z. To be safe, we specify that the remaining values 109-127 ($6D-$7F) translate to themselves.

### Example 2: Those Pesky Zeroes

The Oberheim Xpander's patch names are stored in a way that causes problems for the Librarian. The patch name is 8 characters long, but each character is stored in a two-byte field. Thus, the name appears to have a zero stored in every other byte. When Music-X reads the 16 characters as a name, and tries to print the name on the screen, it stops printing after the first character. Why? Because zero means "end of character string" in most Amiga programming languages, as all C programmers know. How can I print the entire patch name if the first zero effectively cuts off the rest of the name?

To remedy this, I used a Character Map to translate zeroes into space characters (ASCII $20). It leaves all other characters untouched. Here's the Character Map, assuming that the Xpander uses plain ASCII for the rest of its characters (which it does):

\20\20,\01\7F

Now the patch names are displayed correctly, though a blank space appears after each of the 8 characters. We can eliminate the last space at least by claiming that the name length is 15 instead of 16.

Just for fun, let's modify our Char Map to translate capital letters into small letters. The Xpander uses only capital letters internally. On the ASCII table, capital letters are found in positions 65-90 ($41-$5A), so we isolate this range:

\20\20,\01\40,AZ,\5B\7F

and then translate the letters to lower case:

\20\20,\01\40,az,\5B\7F

### Example 3: Nybbling at the Yamaha SPX-90

The Yamaha SPX-90 digital effects unit stores its 16-character patch name in 32 bytes of data. Each character is represented as a two-byte quantity, with only the least-significant 4 bits used in each byte. For example, suppose the first two bytes of the patch name are 4 and 5. In binary, these numbers are 00000100 and 00000101. Extract the least-significant 4 bits in each to get 0100 and 0101. Paste them together to get 01000101. This is the number $45 which is the character 'E' on the ASCII table.

Anyway… can we make Music-X extract this patch name? Turning to the manual, we find that Music-X can indeed understand this "nybblized" data (a nybble is half a byte) by using the Y data variable (page 384). It can grab two bytes, extract their least-significant nybbles, and join them together into a single byte—exactly what we need. Hooray!

Unfortunately, our success is short-lived. Music-X assumes that the first byte arriving is the least-significant, and the second is the most-significant. This is exactly the opposite of what the SPX-90 sends! In other words, in our $45 example above, Music-X will interpret this as the number $54 (the letter 'T'). Our method won't work.

In fact, our method has a second problem. If we use the Y variable to extract nybbles, this changes the actual data stored by Music-X, and therefore alters the value of our computed checksum (K variable)! In order to make this work, we would have to disable the checksum handling in our protocol, and treat the checksum as an ordinary data byte.

As of this writing, I still have not managed to get Music-X to understand SPX-90 patch names. If anybody else can figure out a method, please contact me.

### Conclusions

I hope that this article has made protocols less mysterious for you. I've now written protocols for about 10 different instruments (including some fairly old synthesizers), and the process gets easier each time. Don't be afraid to experiment: you can't break anything by writing an incorrect protocol. As with any computer data, however, make sure to back up your patches in some other way before experimenting, in case you accidentally overwrite something. Good luck!

### ADDENDUM: A Few Bug You May Encounter (Music-X 1.1)

I encountered a few bugs while using the Librarian and Protocol Editor. If you are editing an existing library's protocol, and you Load… a new protocol and return to the Librarian Page, the old protocol's name will still be listed under the "—Format—" label. In fact, if you now re-enter the Protocol Editor, the new protocol's name has been changed to the old name (although the rest of the new protocol is OK).

If you choose Load… in the Librarian, and then click on CANCEL in the requestor, your current library disappears from the screen (and is replaced by "No Page"). I consider this a bug because "CANCEL" should return the program to the exact state it was in before the requestor appeared. To bring back your library, use the Set Display command in the Edit menu.

If you are in the Protocol Editor and you choose Load… or Save… from the File menu, sometimes the file requestor's Directory gadget contains an incorrect value. This causes the message "—Not a valid directory—" to be displayed in the requester. If you examine the directory name, you will find that the end of the directory name has been replaced by the PROTOCOL's name! This happens if your directory name is longer than 32 characters.

I have managed to put Music-X into an infinite loop (although its menu bar keeps working) by choosing Load in the Librarian when there is already a library loaded. The problem is intermittent and may be related to the above directory bug.

### About the Author

Daniel Barrett has been working in electronic and computer music since 1979, and was the moderator of the Internet Music-X electronic mailing list (now defunct). He is currently a Ph.D. candidate in computer science at the University of Massachusetts, and may be reached by electronic mail as barrett@cs.umass.edu (Internet) or >internet:barrett@cs.umass.edu (Compuserve). Special thanks go to Mike Metlay for the use of his two Prophets.

# AudioProbe

*by Jim Olinger*

Since the Amiga has the best built-in sound capabilities of any personal computer, it's rather surprising how little attention Amiga sound programming has received. This may be so because most people are visually oriented. Humans are usually aware of what they are seeing. Sound is often subliminal. Hearing is a more subtle, primal sense.

Or maybe it's just because it's so easy to play sampled sounds on the Amiga that few programmers have explored the alternatives, which call for diving deeply into Exec and I/O devices. Most games seem to use sampled sounds exclusively.

There are two general techniques for producing sounds with a computer: sampling and synthesis. Each has advantages and disadvantages.

Sampling consists of playing back digitized sounds. The most realistic musical instrument sounds and most impressive sound effects are usually samples. The Amiga's advanced audio chips usually make sample playback a "fire and forget" procedure. The CPU can pass the sampled sound data and play instructions to the audio hardware and go on about its other business.

Since samples are essentially audio recordings, a sample user faces all the problems of an audio engineer. The sound is usually recorded, digitized and then manipulated with sample editing software before it is ready to use. It's no wonder that most games include "sound designer" in the credits! Samples also tend to be large; 16 to 32K per sample is common.

Synthesis is the process of manipulating relatively simple waveforms. The waves don't take up nearly as much memory as samples, but more attention from the CPU may be required. Synthesized sounds aren't as realistic as samples (although you might ask what a "real" phaser or hyperspace jump sounds like), but a synthesized sound might be perfect for a cockpit alarm or for an instrument in a musical soundtrack.

Both sound generation techniques are valuable, and they are frequently used together. "Star Wars" is an excellent example. Every environment, such as Luke's planet, Obi-Wan Kenobi's home and the Death Star, had a characteristic background sound produced by simple analog synthesizers. Starship engines were derived from a recording of the Goodyear blimp. Darth Vader's voice was produced by radical filtering and electronic processing of James Earl Jones' voice. The unnaturally regular hissing of Darth's breath was another synthesizer sound. R2-D2's voice was a wild mixture of human voices, puppy cries and synthesizers. And so on.

The accompanying program, AudioProbe1, was written to explore the application of analog synthesizer concepts to Amiga sound generation. It dynamically manipulates pitch and volume (the two simplest sound parameters). It is oriented toward processing simple waves, but it will also work with sampled sounds. In fact, most programmers will probably use the AudioProbe1 techniques to increase the utility of individual samples by dynamically modifying the sample to produce different sound effects.

Before we get into the program, let's examine some basic synthesizer concepts and the Amiga audio hardware.

## Analog Synthesizer Basics

The first electronic instrument was the Telharmonium, which was intended to send music to subscribers over telephone lines. It was invented in the 1890s. This service never took off, but the Telharmonium was the basis for the Hammond organs, developed in the 1930s. Several other electronic instruments were invented in the early decades of this century.

"Classical" electronic music emerged after World War II. It was based on manipulating sounds with the newly available tape recorders, and was actually closer to modern samplers than to the analog synthesizers which appeared a few years later.

The modern commercial synthesizer was developed in the late 1960s by Robert Moog. The first instruments filled a significant part of a room. Sounds were "programmed" by connecting individual components with a tangle of patch cords. The early "Moogs" (practically a synonym for "synthesizer" at the time) were custom-built, temperamental, and very expensive.

The first affordable synthesizer was the Mini-Moog (produced from 1970 to 1981), which combined the large instrument's most often-used components into a suitcase-sized package. It contained an organ-style keyboard, voltage-controlled oscillators (VCOs), a noise generator, a mixer, a voltage-controlled filter (VCF), a voltage-controlled amplifier (VCA), and two envelope generators. Figure 1 is a block diagram of a generalized analog synthesizer voice, similar to a Mini-Moog. A Mini-Moog could play only one note at a time. Modern synthesizers consist of a number of these voices, allowing several notes to be played simultaneously.

Many musicians still prize their Mini-Moogs, which can produce sounds that are difficult to recreate on any other synthesizer.

The Mini-Moog sound sources are three voltage-controlled oscillators, which produce simple, but harmonically rich, waveforms, and the noise generator, a circuit which creates a "hiss," like radio static. The number of complete wave cycles the oscillators generate each second determines the pitch of the sound. Noise, being a set of random, non-repeating frequencies, has no distinguishable pitch.

# Sound Synthesis Experiments in Modula-2

The waves from the VCOs are the basis of most instrument sounds while noise is useful for wind or surf effects, percussion, gunshots, or adding "breath" to flute sounds.

Figure 2 shows some common synthesizer waveforms. Compare them to the acoustic instrument waveforms in Figure 3.

The mixer combines the sound sources for processing by the filter and amplifier.

The voltage-controlled filter is a "super tone control." It passes all frequencies below a certain "cut-off" point and removes all frequencies above that point. Filtering a waveform changes its shape. Since the waveshape determines the timbre of a sound, the filter is the primary timbre modifier.

The voltage-controlled amplifier controls the final volume of the sound.

One of the oscillators could be switched from voltage-controlled operation in the audio-range (20-20,000 Hertz) to low frequency output (0.2-20 Hertz) which could be directed to the VCOs to produce vibrato and/or to the VCF to create periodic timbre changes.

The envelope generators output voltages which vary over time. One envelope generator is connected to the filter, allowing dynamic timbre variations. The other envelope generator controls the amplifier.

Figures 4a and 4c are drawings of the waveforms produced by an organ and guitar playing three staccato (detached) notes, with high, medium ,and low pitches. Figures 4b and 4d are graphs of volume changes over the time the notes are being played and were obtained by connecting the highest volume points of the waveforms. These graphs are called "envelopes" and they represent the characteristic "shape" of an instrument note.

An organ note starts playing at full volume as soon as a key is pressed, continues playing at that volume as long as the key is held down, and ends abruptly when the key is released. A guitar note starts with maximum volume when the string is plucked and the volume decreases smoothly and fairly quickly.

A trumpet note (Figures 4e and 4f) begins with a quick smooth volume increase, then drops back to a steady level. When the player stops blowing, the volume quickly, but not instantly, decreases to zero.

The envelope is an extremely important part of an instrument's characteristic sound. One way of producing unique sounds is playing an instrument's waveform with the envelope of a different instrument. For example, a trumpet waveform played with an organ or guitar envelope will sound more like an organ or guitar then a trumpet.

A trumpet envelope is the model for the ADSR (Attack, Decay, Sustain, Release), the most popular envelope generator (Figure 5). When a key is pressed, the output voltage climbs from zero to maximum at a rate controlled by the "attack time" control. After reaching maximum, the voltage drops to the "sustain level" over the "decay time." It remains at the sustain level until the key is released and falls to zero over the "release time."

The AudioProbe1 program's ADSRs have an additional parameter, "sustain time," which controls the amount of time the envelope remains at the sustain level.

Oscillators, filters, and amplifiers had been used in "electronic music" for over 20 years when the Mini-Moog was developed. They were everyday radio equipment. "Voltage control" was the key difference. The earlier components had to be manipulated manually. For example, a musician had to turn an oscillator's frequency knob to change pitch. Playing a scale required recording each individual note and then splicing the tape.

Voltage control changed all that. Voltage from the keyboard controlled oscillator frequency. Voltages from the envelope generators shaped the response of the filter and amplifier. This gave the musician real-time control of pitch, timbre, and volume.
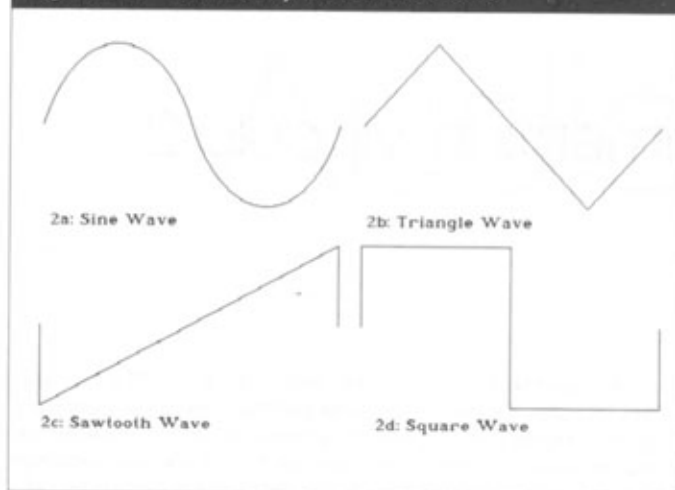
The Mini-Moog and the ARP Odyssey, a similar synthesizer built by Moog's principal competitor, ARP Instruments, moved electronic music from the recording studio into the world



**Figure One**   Generalized Synthesizer Voice
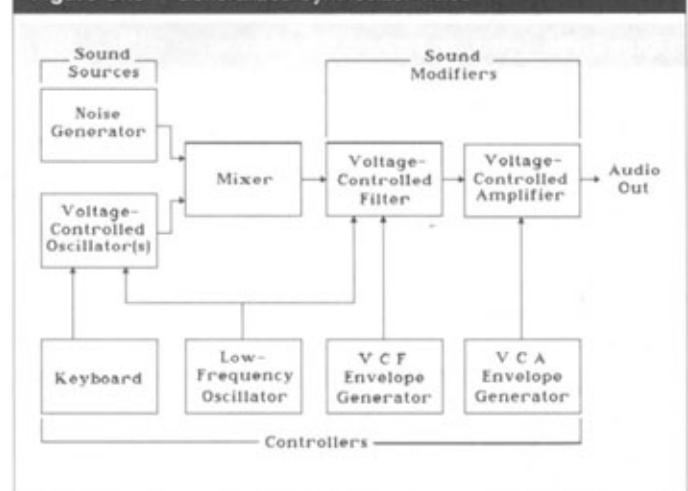
**Figure Two** Common Synthesizer Waveforms

2a: Sine Wave

2b: Triangle Wave

2c: Sawtooth Wave

2d: Square Wave

of live performance. Most modern synthesizers still use the sound-generating techniques devised for these pioneering instruments.

## Computer Synthesis

Many analog synthesis concepts relate directly to computer synthesis.

A computer can't produce analog waveforms directly. Instead, a waveform is represented by a string of numbers, usually called a wavetable. To create a wavetable, one must divide the time axis of a graph of the waveform into equal segments. The points of the waveform at each time interval are called samples.

To play a sound, the computer scans the wavetable, sending each sample to a digital-to-analog converter (DAC). The DAC output is sent to an amplifier, then to a loudspeaker. The frequency of the output waveform is determined by the speed at which the wavetable is scanned.



**Figure Three** Acoustic Instrument Waveforms

3a: Flute

3b: Violin

3c: Mellow Clarinet

3d: Bright Clarinet

3e: Trumpet

3f: Timpani
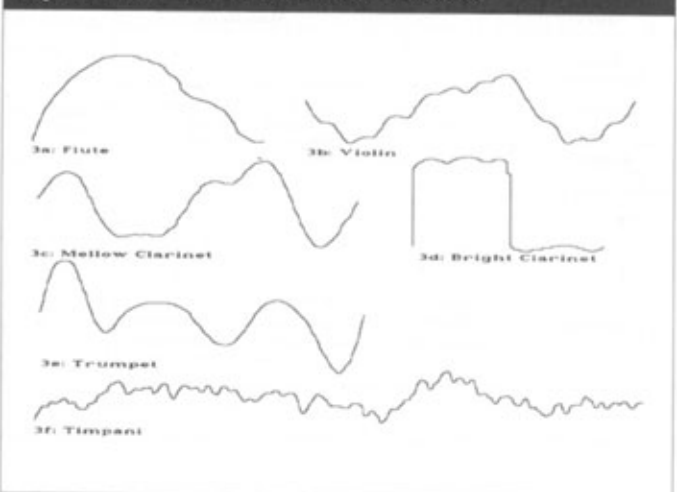
Analog oscillators contain circuits to produce a limited number of waveforms, such as sawtooth, square, and pulse waves. The raw waveforms are good for producing some pipe organ timbres, but are too harshly "electronic" for most musical applications. The filter is used to tailor the waveform to create more pleasing timbres.

Computers usually don't have voltage-controlled filters to alter waveforms. Instead, the wavetable can be loaded with any waveform. One analog feature is lost: it's difficult to change a computer waveform dynamically. This can be done with a "double buffering" technique where a new wavetable is calculated while another wavetable is being played. The second wavetable is then played while the first is recalculated. A detailed discussion of double buffering will have to wait for another article.

Voltage-controlled amplifiers can be computer-simulated by adjusting the amplitude of the samples in the wavetable or by controlling the digital-to-analog converter's output level. The second method is preferred, if the hardware supports it. To minimize distortion, the wavetable should use the maximum available sample range and volume should be controlled by adjusting the DAC output. Envelope generators, low frequency oscillators, and other controllers are simulated in software.

## Amiga Audio Hardware

The Amiga has four independent hardware audio channels ("voices"). Each voice contains a direct memory access (DMA) channel, an eight-bit digital-to-analog converter and an amplifier. Channels 0 and 3 are connected to the left stereo output while channels 1 and 2 are connected to the right output jack.

Complex audio effects can be created by using one audio channel to modulate the amplitude or frequency of another channel. We will concentrate on DMA playback of wavetables. Each audio channel is controlled by registers containing control data, the wavetable starting address, the wavetable length, the output volume, and the sample playback period. The output volume ranges from 0 (silent) to 64 (maximum).

The playback period specifies the number of system clock ticks to wait before sending another sample to the DAC. One system tick is 0.279365 microseconds. The DAC requires 124 ticks (34.642 microseconds) to perform a conversion. Therefore, the minimum period value is 124 ticks. Since the frequency of a sound is the number of complete waveform cycles per second, the output frequency is determined by the wavetable length and the sample period. Note that frequency increases as period decreases. Here is the equation for calculating the period:

$$P = \frac{1\ cycle}{L\ samples} \times \frac{1\ second}{F\ cycles} \times \frac{1,000,000\ micro\ secs}{1\ second} \times \frac{1\ interval}{0.279365\ micro\ secs}$$

This reduces to:

$$P = \frac{3579546.4\ intervals}{L * F\ sample}$$

```
P = period in system clock ticks.
L = wavetable length in samples
F = frequency in cycles per second (Hertz)
```

The audio software is implemented as a standard Amiga input/output device. It is controlled by standard I/O device commands, as described in Part I Chapter 4 (I/O) of the *ROM Kernel Manual*.

### The AudioProbe1 Program

AudioProbe1 consists of several modules, written in Benchmark Modula-2. AP1Voices contains all the procedures for allocating, playing, modifying, and releasing voices. AP1Envelopes implements volume and frequency envelope generators for each voice. AP1Waves calculates several basic synthesizer waves, noise and user-specified waveforms. Noise is the basis for a large number of sound effects. The AudioProbe1 main program is primarily control logic and procedures for demonstrating features of the other modules.

AudioProbe1 can read sampled sounds from disk, but it also generates simple synthesizer waveforms. It includes numerous procedures for experimenting with waveforms. The AudioProbe1 procedures for allocating, playing, stopping, and releasing voices are derived from the program in Len White's article "Digitized Sound Playback in Modula-2" (*Amazing Computing*, May 1989).

The most important addition is the ability to modify the frequency and volume of sounds while they're being played. When I started writing the program, I expected this to be simple. It wasn't. An obvious way of changing frequency or volume is to play a voice, stop it, change the desired parameters, and start the altered voice. There is a distinct "pop" when the old voice is turned off. It's impossible to make smooth changes with this method.

A voice is controlled by sending an I/O request to the audio device with the voice parameters in an IOAudio record. How about changing the parameters in this record while the voice is playing and sending another I/O request? This causes two problems. First, it doesn't work. The frequency and volume stay the same. Even worse, the computer crashes when the voice is turned off. The "I/O" chapter of the *ROM Kernel Manual* warns, "I/O request blocks, once issued, must not be modified or reused until they are returned to your control by Exec." Exec doesn't release the request block until the voice has stopped playing.

Each of the four voices is allocated by opening a communications port, creating an I/O request, opening a copy of the audio device for the channel, and sending an "allocate" command to the device.

To control a playing voice, one must create another I/O request, with its own communications port and copy of the audio device. The "period" and "volume" fields of this "control" I/O request are set to the desired values and the "unit," "allocation key," and "data" fields are loaded with data from the I/O request block of the channel to be modified. This "change period/volume" command is sent to the control I/O request's copy of the audio device.

The device procedure "BeginIO()," rather than "DoIO()," is used to play and alter voices because "BeginIO()," unlike "DoIO()," doesn't reset the device flags in the I/O Request.



**Figure Four**   Instrument Volume Envelopes

4a: Staccato organ notes

4b: Volume envelope of staccato organ notes

4c: Staccato guitar notes

4d: Volume envelope of staccato guitar notes

4e: Staccato trumpet note

Volume
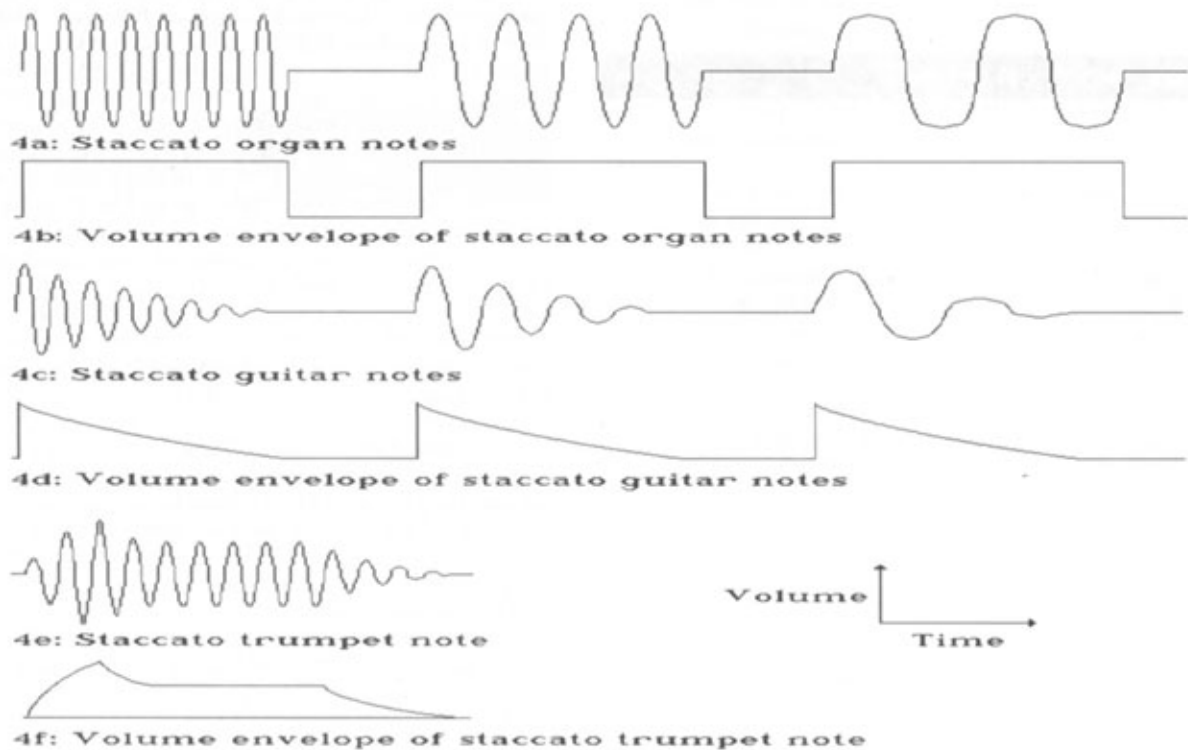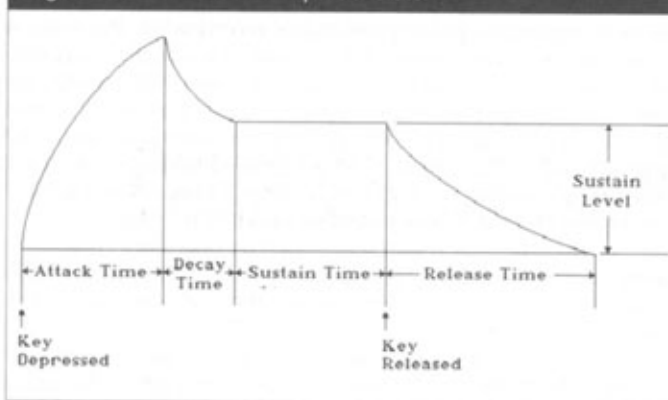
Time

4f: Volume envelope of staccato trumpet note

## Figure Five ADSR Envelope Generator



## Operating AudioProbe1

Since AudioProbe1 is a program for experimenting with sounds, it has a simple character-based user interface to allow maximum flexibility (Figure 6). If AudioProbe1 is run from a console window, like the window created for the "Run Main Module" function of the Benchmark editor, the program is controlled by single keypress commands. If it is run from the Shell or CLI, the "Return" key must be pressed after the command key to send the keystrokes to the program.

"Select Waveform" calls procedures which create simple wavetables. It also can load a sampled sound from disk. When a waveform is selected, it is played for two seconds at "Sample Volume". "Load Sample" is for "raw" samples. It will not work correctly with IFF format samples.

## Figure Six AudioProbe1 Controls

```
======================= AudioProbe1 ==========================

WaveForm: _____

Frequency: _____
Modulation(%): ___ Sustain Level(%): ___
Attack: _____ Decay: _____ Sustain: _____ Release: _____

Volume: __
Modulation(%): ___ Sustain Level(%): ___
Attack: _____ Decay: _____ Sustain: _____ Release: _____

Sample: Frequency: _____ Volume: __ Cycles: ___

W-Load Waveform 1-Sine 2-Sine32 3-Sine64 4-Triangle
5-Flute 6-Square 7-Clarinet 8-Sawtooth 9-Noise
F-Set Frequency V-Set Volume E-Frequency Envelope G-Volume Envelope
P-Set Sample Frequency L-Set Sample Volume T-Set Sample Cycles
B-Play Broken Chord C-Play Chord N-Play Note
S-Play Sample Z-Play Siren Q-Quit


_____ Frequency/Volume Envelope _____
Modulation(%): ___ Sustain Level(%): ___
Attack: _____ Decay: _____ Sustain: _____ Release: _____

M-Modulation L-Sustain Level
A-Attack Time D-Decay Time S-Sustain Time R-Release Time
C-Play Chord N-Play Note E-Exit
```

Free sampled sounds can be found on public domain disks and bulletin boards. A number of companies listed in the "Music" section of *AC's Guide to the Amiga* sell disks of sounds. Some games, such as *Falcon*, have sound files which can be used for experiments. Of course, it's probably illegal and certainly unethical to use other people's game sounds in your own commercial product. Numerous audio digitizers, some costing less than $100, are available. Programs which will generate "samples" also exist. The AudioProbe1 "noise" waveform is an example of a computed "sample."

Two of the available waveforms, "Clarinet" and "Flute," were obtained by digitizing drawings of the waveforms from a music book and reading evenly-spaced points with the co-ordinate feature of DeluxePaint III. Wave sample values also could be determined by drawing the wave on graph paper.

The waves "Sine," "Sine32," and "Sine64" demonstrate the differences in timbre between the same waveform represented by 20, 32, and 64 sample values.

"Set Frequency" controls the playback period. For short waveforms, frequency directly corresponds to the pitch of the output sound. However, since pitch is determined by both period and wavetable length, and the minimum period is 124 system clock ticks, long wavetables can't be played at high frequencies. For example, the length of the longest wavetable that can be played at 440 Hertz ("middle A") is 64 bytes. This length is sufficient for simple single-cycle waves.

If the period is set to less then 124, the audio hardware won't have enough time to retrieve the next data sample and the previous sample will be reused. This produces unexpected audio effects. All the frequency-related procedures in the AP1Voices module avoid this problem by setting any period values less than 124 to 124 and writing an error message. I call this adjustment "period clipping."

Sampled sounds, with wavetable lengths of 16K to 32K, contain many wave cycles, making the exact playback pitch difficult to calculate. When playing long wavetables, a frequency under 1 Hertz should be used. Attempting to play a wavetable longer than 28,866 bytes at 1 Hertz will cause period clipping.

"Set Volume" adjusts the playback volume from 0 (silent) to 64 (full). Both frequency and volume are "base" values, which can be increased or decreased by the envelope generators. There are two independent envelope generators. One modifies frequency while the other modifies volume.

"Modulation" is the maximum variation from the base value, expressed in percent. Positive and negative modulation amounts are allowed. The maximum volume modulation amount is -100% to +100%. A frequency modulation amount of +100% doubles the frequency, raising the pitch by one octave, while -100% halves the frequency, dropping the pitch by one octave. Frequency modulations of several hundred percent are possible.

"Attack Time," "Decay Time," "Sustain Level," "Sustain Time," and "Release Time" control the envelope shape. "Set Sample Frequency" and "Set Sample Volume" adjust frequency and volume for playing samples without using the envelope generators. "Set Sample Cycles" controls the number of times a sample will be repeated. These functions are used mainly for auditioning new samples.

"Play Broken Chord" is a stereo demonstration. It starts each voice individually, building up to a four-note chord, then removes the notes individually. The envelope generators are not used.

"Play Chord" plays a four-note chord. Frequency and volume can be modified by the envelope generators. If the frequency envelope is applied, the result is similar to a slide guitar chord. A major chord, containing the first, third, fifth, and octave notes of a major scale, is produced. The root note of the chord is determined by the "Frequency" parameter.

"Play Note" plays a single note, controlled by the "Frequency" and "Volume" parameters and the envelope generators.

The notes used in the chord experiments are computed using the relationships of the equal-tempered musical scale. In this system, the frequency of each note is the twelfth root of 2 (1.059463) times the frequency of the previous note. Figure 7 shows two octaves of notes in standard concert tuning, where "middle A" is 440 Hertz. The lowest scale note is set from the "Frequency" parameter.

"Play Siren" demonstrates volume and frequency changes without using the envelope generators.

"Play Sample" plays the current waveform, controlled by the "Sample Frequency," "Sample Volume," and "Sample Cycles" parameters. This is primarily an "audition" function. The envelope generators are not used.

## Experiments

Here are a few pointers before we get into the experiments.

In the unlikely event that you haven't already hooked your Amiga's audio outputs to a stereo, I strongly recommend you do so before experimenting with AudioProbe1. The distortion from the tiny amplifier and speakers in your video monitor can give you extremely misleading results. You will also miss the stereo effects if you use a mono monitor.

AudioProbe1 is almost as complex as a Mini-Moog, which, to the beginner, is complex indeed. Many functions interact, often in ways which aren't immediately obvious. The Mini-Moog provides instant feedback. When a switch is flipped or a knob is turned, something (usually) happens instantly. AudioProbe1 is controlled by pressing keys to select menu items and typing numbers. This is more abstract than operating controls directly.

To avoid becoming disoriented, experiment with changing one parameter at a time. Try many values of each parameter and many combinations of parameters. Don't be afraid to try strange combinations. In synthesis, many impressive and useful sounds have been discovered by accident.

If something unexpected happens, compare the relationships of all the elements. One common problem is envelope conflicts. If a sound has a long frequency envelope and a short volume envelope, the volume envelope will cut the sound off before the frequency envelope is finished.

The "Chord" and "Siren" functions have maximum pitches an octave above the "Frequency" parameter ("Frequency" * 2). The Frequency Envelope Generator can increase

pitch indefinitely, based on the "Modulation Amount" parameter (+100% modulation = 1 octave, +200% modulation = 2 octaves, etc.). The maximum frequency before period clipping occurs is calculated by the equation: Maximum Frequency = 28867 / WaveLength. The following table gives the frequencies for the built-in AudioProbe1 waveforms.

| Waveform Name | Waveform Length | Max Note Frequency | Max Chord Frequency |
|---|---|---|---|
| Sine | 20 | 1443 | 721 |
| Sine32 | 32 | 902 | 451 |
| Sine64 | 64 | 451 | 225 |
| Triangle | 32 | 902 | 451 |
| Flute | 62 | 465 | 232 |
| Square | 20 | 1443 | 721 |
| Clarinet | 52 | 555 | 277 |
| Sawtooth | 32 | 902 | 451 |
| Noise | 4096 | 7 | 3 |

Setup: Set "Sample Volume" to 64. Select "Sawtooth" wave. While the "Audition" note is playing, adjust the volume on your stereo to a comfortable level.

Waveform Resolution: Set "Sample Volume" to 64. Select the "Sine," "Sine32," and "Sine64" waveforms and compare their timbres. Notice that Sine32 and Sine64 are practically indistinguishable while Sine has a distinct high-pitched whine added to the fundamental sound. This suggests that 32 samples is the optimum sample length for simple single-cycle waveforms.

Waveform Timbres: Set "Sample Volume" to 64. Compare the timbres of waves with similar waveforms (see Figures 2 and 3) such as "Sine32," "Triangle," and "Flute" or "Square" and "Clarinet" (named "Bright Clarinet" in figure 3d).

Stereo Voices: Select any waveform. Set "Frequency" to avoid period clipping. Set "Volume" to 64. Play "Broken Chord". Notice how the individual notes appear in alternating speakers.

Siren: Select any waveform. Set "Frequency" to avoid period clipping. Play "Siren." Be sure to try this with all waveforms, using numerous frequencies, including extremely low frequencies.

Volume Envelope: Select the "Sine32" waveform. Set "Frequency" to 440 and "Volume" to 0. Set "Frequency Envelope Modulation" to 0%. "Volume Envelope" parameters are Modulation = 100%, Sustain Level = 50%, Attack = 0.5, Decay = 0.5, Sustain = 1.0 and Release = 0.5. Play "Note" and "Chord." Notice that the volume builds to full, drops back to half, sustains, and finally decreases to zero. Vary the Attack, Decay, Sustain, Sustain Level, and Release parameters and observe the change. Set "Volume" to 16 and "Volume Envelope Modulation" to 75%. Play "Note." This time, the volume starts at 16 and builds.

Frequency Envelope: Select the "Sine32" waveform. Set "Frequency" to 110 and "Volume" to 64. "Frequency Envelope" parameters are Modulation = 100%, Sustain Level = 50%, Attack = 0.5, Decay = 0.5, Sustain = 1.0 and Release = 0.5. Set "Volume Envelope Modulation" to 0%. Play "Note" and "Chord". Vary the "Frequency Envelope" parameters. Try "Frequency Modulation" = +200% and -200%.

Volume and Frequency Envelopes: Vary the "Frequency Envelope" and "Volume Envelope" parameters with non-zero modulation amounts.

**Figure Seven**
Equal-Tempered Scale Frequencies

| Note | Frequency |
|---|---|
| A | 440.0 |
| A# | 466.2 |
| B | 493.9 |
| C | 523.3 |
| C# | 554.4 |
| D | 587.3 |
| D# | 622.3 |
| E | 659.3 |
| F | 698.5 |
| F# | 740.0 |
| G | 784.0 |
| G# | 830.6 |
| A | 880.0 |
| A# | 932.3 |
| B | 987.8 |
| C | 1046.5 |
| C# | 1108.7 |
| D | 1174.7 |
| D# | 1244.5 |
| E | 1318.5 |
| F | 1396.9 |
| F# | 1480.0 |
| G | 1568.0 |
| G# | 1661.2 |

Loony Tunes: Select the "Sine" waveform. Set "Frequency" to 110 and "Volume" to 0. "Frequency Envelope" parameters are Modulation = 200%, Sustain Level = 100%, Attack = 1.0, Decay = 0.0, Sustain = 1.0 and Release = 0.5. "Volume Envelope" parameters are Modulation = 100%, Sustain Level = 100%, Attack = 1.0, Decay = 0.0, Sustain = 1.0 and Release = 0.5. Play "Chord". Does this sound a bit like the beginning of a certain cartoon?

Noise Experiments: Noise is a basic synthesizer waveform that is more like a sample than a simple waveform. The following experiments demonstrate some of the ways noise can be used.

Jet Takeoff (heard from ground): Select "Noise" waveform. For a single-engine jet, set "Frequency" to 0.1 and "Volume" to 0. "Frequency Envelope" parameters are Modulation = 200%, Sustain Level = 100%, Attack = 4.0, Decay = 0.0, Sustain = 12.0 and Release = 0.0. "Volume Envelope" parameters are Modulation = 100%, Sustain Level = 100%, Attack = 4.0, Decay = 0.0, Sustain = 6.0 and Release = 6.0. Play "Note." For a multi-engine jet, set "Frequency" to 0.025 and play "Chord." Adjust any or all of these parameters to your own taste.

To increase realism, play one or two voices with the "Noise" waveform and add a high pitched whine by playing additional voices with the "Sine" or "Triangle" waveforms. This will require additional programming (see "Improvements").

Jet in Flight: Select "Noise" waveform. For a single-engine jet, set "Frequency" to 0.1 and "Volume" to 32. Set both "Frequency Modulation" and "Volume Modulation" amounts to zero. Control the time the sound plays by adjusting the "Sustain Time" for either the Frequency or Volume Envelope Generator. Play "Note." For a multi-engine jet, set "Frequency" to 0.05 and play "Chord."

## Improvements

AudioProbe1 wasn't written as a music program, but music entry and editing capabilities could be added. However, although the Amiga has very good sound-producing capabilities (for a computer), you would probably get better results by buying a synthesizer, MIDI interface, and sequencer software if your primary interest is music.

AudioProbe1 doesn't provide the instant feedback of a Mini-Moog or ARP Odyssey synthesizer. This could be corrected by creating an Intuition-based control panel, which could be very useful for sound design experiments.

Data for arbitrary waveforms is created by manually determining the sample values. The wave could be entered by drawing it on the screen with the mouse. A procedure to save wave data to disk could be added. Procedures for reading and writing IFF sound files could also be added.

The ADSR envelope generator has the smallest number of parameters that can represent a complex instrument envelope, such as a trumpet envelope. It was popular with early synthesizer designers because it provided considerable flexibility with a minimum of components. With software envelope generators, we can have any number of stages and levels. In fact, the AP1Envelopes module is written so that the only modifications required to change the number of envelope stages are changing the limits of the "Time" and "Level" arrays in the "EnvelopeRec" data structure and adding statements to the "SetEnvelope()" procedure to initialize the additional stages.

For simplicity, AudioProbe1 uses the same waveform, frequency, volume, and envelope generator parameters for all four voices. Since the hardware for each voice is independent, all the controls for each voice could also be independent.

One distinct problem with the AudioProbe1 envelope generators is that they take over control of the program until the envelope cycle is complete. This might be fine for short sound effects, but it could interfere with other activities, such as screen redraws, if a sound has a long envelope. One way of dealing with this problem would be to move envelope generator updates into the normal processing loop. Since the Amiga is a multi-tasking machine, a better solution would be to create a separate task to operate the envelope generators.

### The End (Of The Beginning)

This has been a brief introduction to sound synthesis. As with all skills, expertise in synthesis requires continued study and practice. Develop your ears. Listen to all the sounds around you and consider how they could be synthesized. Analyze the basic components of a sound and determine how they can be manipulated to produce the effect you want. Be alert for happy accidents. Many discoveries have been made on the way to something else.

An amazing and wonderful world of sound awaits you.

### Bibliography

*Amiga Hardware Manual*:
Chapter 5 - Audio Hardware, pp 5-1 to 5-31.

*Amiga ROM Kernel Manual*: Part III
Chapter 1 - Audio Device, pp 3-1 to 3-28.

*Amiga ROM Kernel Manual*: Part III
Chapter 2 - Timer Device, pp 3-29 to 3-41.

*Learning Music With Synthesizers*. David Friend, Alan R. Pearlman, Thomas D. Piggott; Hal Leonard Publishing Corporation, 1974.

"Digitized Sound Playback in Modula-2", Len A. White, *Amazing Computing* (May 1989), pp 45-47.

**Listing One**    AudioProbe1.mod

```
MODULE AudioProbe1;

(*  _____
    Program: AudioProbe1
    Purpose: Amiga Sound Experiments
    Author : Jim Olinger (Armadillo Computing)
    File   : Amazing:AudioProbe1.mod
    Date   : 01-Aug-91  20:40
    _____ *)

FROM AP1Envelopes IMPORT
    (* type *) EnvelopeRec,
    (* proc *) DeleteEnvelopeTimer, EnvelopeTimerCreated,
               Pause, PlayEnvelopes, SetBaseFrequency,
               SetBaseVolume, SetFrequencyEnvelope,
               SetVolumeEnvelope;

FROM AP1Voices IMPORT
    (* proc *) PeriodCalculated, PlayVoice,
               PlayVoicePerCyc, ReleaseVoices,
               ReleaseWaveBuffer, SetVoiceFrequency,
               SetVoiceVolume, SetVoiceWave, StopVoice,
               VoicesAllocated, WaveBufferAllocated,
               WaveRead;

FROM AP1Waves IMPORT
    (* proc *) MakeNoiseWave, MakeSawtoothWave,
               MakeSineWave, MakeSquareWave,
               MakeTriangleWave, MakeWaveFromArray;

FROM ASCII IMPORT
    (* const *) LF;
```

```
FROM AudioDevice IMPORT
    (* const *) Left0, Left1, Right0, Right1;

FROM InOut IMPORT
    (* proc  *) Read, ReadCard, ReadInt, ReadString,
                WriteCard, WriteInt, WriteString;

FROM RealInOut IMPORT
    (* proc  *) ReadReal, WriteReal;

FROM SYSTEM IMPORT
    (* type  *) ADDRESS;

VAR
    (* wave buffer *)
    waveBuffPtr: ADDRESS;
    waveBuffLen: LONGCARD;
    waveName   : ARRAY [0..128] OF CHAR;

    (* frequency envelope controls *)
    frequency         : REAL;
    freqModAmt        : INTEGER;
    freqAttackTime    : REAL;
    freqDecayTime     : REAL;
    freqSustainLevel  : REAL;
    freqSustainTime   : REAL;
    freqReleaseTime   : REAL;

    (* volume envelope controls *)
    volume            : CARDINAL;
    volModAmt         : INTEGER;
    volAttackTime     : REAL;
    volDecayTime      : REAL;
    volSustainLevel   : REAL;
    volSustainTime    : REAL;
    volReleaseTime    : REAL;

    (* sample playback controls *)
    sampleFrequency: REAL;
    sampleVolume   : CARDINAL;
    sampleCycles   : CARDINAL;

(* * * * *   Playback Procedures   * * * * *)

PROCEDURE CalculateScale(
                root     : REAL;
                VAR scale: ARRAY OF REAL
           );
CONST
    HalfStep = 1.059463; (* twelth root of two *)

VAR
    note: CARDINAL;

BEGIN
    (* calculate equal-tempered scale frequencies *)
    (* by multiplying the previous frequency by  *)
    (*  the twelth root of two                    *)
    scale[0] := root;
    FOR note := 1 TO HIGH(scale) DO
        scale[note] := scale[note - 1] * HalfStep;
    END (*FOR*);
END CalculateScale;

PROCEDURE PlayBrokenChord();

VAR
    scale: ARRAY [0..12] OF REAL;
    vol  : CARDINAL;

BEGIN
    WriteString("Play Broken Chord\n");
    (* calculate scale frequencies *)
    CalculateScale(frequency, scale);
    (* set play volume *)
    IF volume < 16 THEN
        vol := 32;
    ELSE
        vol := volume;
    END (*IF*);
    (* play *)
    PlayVoice(0, scale[00], 0.0, vol);
    Pause(0.5);
    PlayVoice(1, scale[04], 0.0, vol);
    Pause(0.5);
    PlayVoice(2, scale[07], 0.0, vol);
    Pause(0.5);
    PlayVoice(3, scale[12], 0.0, vol);
    Pause(1.0);
    StopVoice(3);
    Pause(0.5);
    StopVoice(2);
    Pause(0.5);
    StopVoice(1);
    Pause(0.5);
    StopVoice(0);
END PlayBrokenChord;
```

```
PROCEDURE PlayChord();

VAR
    scale: ARRAY [0..12] OF REAL;
    idx  : CARDINAL;

BEGIN
    WriteString("Play Chord\n");
    (* calculate scale frequencies *)
    CalculateScale(frequency, scale);
    (* initialize envelopes *)
    FOR idx := 0 TO 3 DO
        SetVolumeEnvelope(
            idx,
            TRUE,
            volModAmt,
            volAttackTime,
            volDecayTime,
            volSustainLevel,
            volSustainTime,
            volReleaseTime
        );
        SetFrequencyEnvelope(
            idx,
            TRUE,
            freqModAmt,
            freqAttackTime,
            freqDecayTime,
            freqSustainLevel,
            freqSustainTime,
            freqReleaseTime
        );
    END (*FOR*);
    (* play chord - first, third, fifth, octive *)
    SetBaseFrequency(0, scale[00]);
    SetBaseFrequency(1, scale[04]);
    SetBaseFrequency(2, scale[07]);
    SetBaseFrequency(3, scale[12]);
    SetBaseVolume(0, volume);
    SetBaseVolume(1, volume);
    SetBaseVolume(2, volume);
    SetBaseVolume(3, volume);
    PlayEnvelopes();
END PlayChord;


PROCEDURE PlayNote();

BEGIN
    WriteString("Play Note\n");
    SetVolumeEnvelope(
        0,
        TRUE,
        volModAmt,
        volAttackTime,
        volDecayTime,
        volSustainLevel,
        volSustainTime,
        volReleaseTime
    );
    SetFrequencyEnvelope(
        0,
        TRUE,
```

```
            freqModAmt,
            freqAttackTime,
            freqDecayTime,
            freqSustainLevel,
            freqSustainTime,
             freqReleaseTime
     );
     SetBaseFrequency(0, frequency);
     SetBaseVolume(0, volume);
     PlayEnvelopes();
END PlayNote;


PROCEDURE PlaySiren();

CONST
    FreqSteps   = 400;
    SirenPeriod = 8.0;

VAR
    freqChange: REAL;
    freqDelay : REAL;
    freq      : REAL;
    vol       : CARDINAL;
    step      : CARDINAL;

BEGIN
    WriteString("Play Siren\n");
    (* calculate frequency shift *)
    freqChange := frequency / FLOAT(FreqSteps);
    (* calculate frequency change rate *)
    freqDelay := (SirenPeriod / 4.0) /
                  FLOAT(FreqSteps);
    (* start voice *)
    PlayVoice(0, frequency, 0.0, 0);
    (* fade volume up *)
    WriteString("increasing volume\n");
    FOR vol := 0 TO 64 DO
        SetVoiceVolume(0, vol);
        Pause(0.05);
    END (*FOR*);
    (* bend pitch above base freq *)
    freq := frequency;
    WriteString("freq -> high\n");
    FOR step := 0 TO FreqSteps DO
        SetVoiceFrequency(0, freq);
        freq := freq + freqChange;
        Pause(freqDelay);
    END (*FOR*);
    WriteString("high -> freq\n");
    FOR step := 0 TO FreqSteps DO
        freq := freq - freqChange;
        SetVoiceFrequency(0, freq);
        Pause(freqDelay);
    END (*FOR*);
    WriteString("freq\n");
    (* bend pitch below base freq *)
    freq := frequency;
    freqChange := freqChange / 2.0;
    WriteString("freq -> low\n");
    FOR step := 0 TO FreqSteps DO
        freq := freq - freqChange;
        SetVoiceFrequency(0, freq);
        Pause(freqDelay);
    END (*FOR*);
    WriteString("low -> freq\n");
    FOR step := 0 TO FreqSteps DO
        SetVoiceFrequency(0, freq);
        freq := freq + freqChange;
        Pause(freqDelay);
    END (*FOR*);
    WriteString("freq\n");
    (* fade volume down *)
    WriteString("decreasing volume\n");
    FOR vol := 64 TO 0 BY -1 DO
        SetVoiceVolume(0, vol);
        Pause(0.05);
    END (*FOR*);
    (* stop voice *)
    StopVoice(0);
END PlaySiren;


(* * * * *   Control Procedures   * * * * *)

PROCEDURE ShowEnvelopeParameters(
            modAmt       : INTEGER;
            attackTime   : REAL;
            decayTime    : REAL;
            sustainLevel : REAL;
            sustainTime  : REAL;
            releaseTime  : REAL
         );

BEGIN
    WriteString("Modulation(%): ");
    WriteInt(modAmt, 3);
    WriteString(" Sustain Level(%): ");
    WriteReal(sustainLevel, 1);
    WriteString("\n");
    WriteString("Attack: ");
    WriteReal(attackTime, 1);
    WriteString(" Decay: ");
    WriteReal(decayTime, 1);
    WriteString(" Sustain: ");
    WriteReal(sustainTime, 1);
    WriteString(" Release: ");
    WriteReal(releaseTime, 1);
```

```
    WriteString("\n");
END ShowEnvelopeParameters;


PROCEDURE SetEnvelopeParameters(
            name         : ARRAY OF CHAR;
            VAR modAmt       : INTEGER;
            VAR attackTime   : REAL;
            VAR decayTime    : REAL;
            VAR sustainLevel : REAL;
            VAR sustainTime  : REAL;
            VAR releaseTime  : REAL
         );

VAR
    char: CHAR;

BEGIN
    LOOP
        (* show current parameters *)
        WriteString("————————— ");
        WriteString(name);
        WriteString(" —————————");
        WriteString("\n");
        ShowEnvelopeParameters(
            modAmt,
            attackTime,
            decayTime,
            sustainLevel,
            sustainTime,
            releaseTime
        );
        WriteString("\n");
        WriteString("M-Modulation   ");
        WriteString("L-Sustain Level ");
        WriteString("\n");
        WriteString("A-Attack Time  ");
        WriteString("D-Decay Time   ");
        WriteString("S-Sustain Time  ");
        WriteString("R-Release Time");
        WriteString("\n");
        WriteString("C-Play Chord   ");
        WriteString("N-Play Note    ");
        WriteString("E-Exit");
        WriteString("\n");
        (* process command *)
        Read(char);
        CASE CAP(char) OF
            "A":
                WriteString("Attack Time? ");
                ReadReal(attackTime);
                WriteString("\n");
                IF attackTime < 0.0 THEN
                    attackTime := 0.0;
                END (*IF*);
          |
            "C":
                PlayChord();
          |
            "D":
                WriteString("Decay Time? ");
                ReadReal(decayTime);
                WriteString("\n");
                IF decayTime < 0.0 THEN
                    decayTime := 0.0;
                END (*IF*);
          |
            "E":
                EXIT (*LOOP*);
          |
            "L":
                WriteString("Sustain Level? ");
                ReadReal(sustainLevel);
                WriteString("\n");
                IF sustainLevel > 100.0 THEN
                    sustainLevel := 100.0;
                END (*IF*);
          |
            "M":
                WriteString("Modulation Amount? ");
                ReadInt(modAmt);
                WriteString("\n");
          |
            "N":
                PlayNote();
          |
            "R":
                WriteString("Release Time? ");
                ReadReal(releaseTime);
                WriteString("\n");
                IF releaseTime < 0.0 THEN
                    releaseTime := 0.0;
                END (*IF*);
          |
            "S":
                WriteString("Sustain Time? ");
                ReadReal(sustainTime);
                WriteString("\n");
                IF sustainTime < 0.0 THEN
                    sustainTime := 0.0;
                END (*IF*);
          |
            ELSE
        END (*CASE*);
    END (*LOOP*);
END SetEnvelopeParameters;
```

```
PROCEDURE MakeClarinet();

VAR
   waveArray: ARRAY [0..51] OF INTEGER;

BEGIN
   (* define wave *)
   waveArray[00] := 005;  waveArray[01] := 117;
   waveArray[02] := 120;  waveArray[03] := 122;
   waveArray[04] := 125;  waveArray[05] := 125;
   waveArray[06] := 125;  waveArray[07] := 122;
   waveArray[08] := 122;  waveArray[09] := 120;
   waveArray[10] := 122;  waveArray[11] := 127;
   waveArray[12] := 125;  waveArray[13] := 125;
   waveArray[14] := 127;  waveArray[15] := 127;
   waveArray[16] := 127;  waveArray[17] := 127;
   waveArray[18] := 125;  waveArray[19] := 122;
   waveArray[20] := 122;  waveArray[21] := 122;
   waveArray[22] := 125;  waveArray[23] := 125;
   waveArray[24] := 125;  waveArray[25] := 125;
   waveArray[26] := 105;  waveArray[27] := 005;
   waveArray[28] := 002;  waveArray[29] := 002;
   waveArray[30] := 000;  waveArray[31] := 000;
   waveArray[32] := 000;  waveArray[33] := 002;
   waveArray[34] := 002;  waveArray[35] := 005;
   waveArray[36] := 005;  waveArray[37] := 005;
   waveArray[38] := 007;  waveArray[39] := 007;
   waveArray[40] := 007;  waveArray[41] := 007;
   waveArray[42] := 007;  waveArray[43] := 007;
   waveArray[44] := 007;  waveArray[45] := 005;
   waveArray[46] := 002;  waveArray[47] := 002;
   waveArray[48] := 000;  waveArray[49] := 002;
   waveArray[50] := 005;  waveArray[51] := 007;
   (* make wave *)
   waveName    := "Clarinet";
   waveBuffLen := HIGH(waveArray) + 1;
   waveBuffPtr := MakeWaveFromArray(waveArray);
END MakeClarinet;


PROCEDURE MakeFlute();

VAR
   waveArray: ARRAY [0..61] OF INTEGER;

BEGIN
   (* define wave *)
   waveArray[00] := 000;  waveArray[01] := 017;
   waveArray[02] := 030;  waveArray[03] := 037;
   waveArray[04] := 048;  waveArray[05] := 061;
   waveArray[06] := 063;  waveArray[07] := 071;
   waveArray[08] := 078;  waveArray[09] := 086;
   waveArray[10] := 091;  waveArray[11] := 094;
   waveArray[12] := 099;  waveArray[13] := 104;
   waveArray[14] := 109;  waveArray[15] := 109;
   waveArray[16] := 114;  waveArray[17] := 117;
   waveArray[18] := 117;  waveArray[19] := 122;
   waveArray[20] := 124;  waveArray[21] := 124;
   waveArray[22] := 127;  waveArray[23] := 127;
   waveArray[24] := 127;  waveArray[25] := 127;
   waveArray[26] := 127;  waveArray[27] := 127;
   waveArray[28] := 127;  waveArray[29] := 122;
   waveArray[30] := 122;  waveArray[31] := 122;
   waveArray[32] := 119;  waveArray[33] := 117;
   waveArray[34] := 114;  waveArray[35] := 112;
   waveArray[36] := 107;  waveArray[37] := 099;
   waveArray[38] := 096;  waveArray[39] := 091;
   waveArray[40] := 081;  waveArray[41] := 071;
   waveArray[42] := 068;  waveArray[43] := 066;
   waveArray[44] := 066;  waveArray[45] := 063;
   waveArray[46] := 061;  waveArray[47] := 061;
   waveArray[48] := 058;  waveArray[49] := 053;
   waveArray[50] := 050;  waveArray[51] := 048;
   waveArray[52] := 043;  waveArray[53] := 033;
   waveArray[54] := 027;  waveArray[55] := 020;
   waveArray[56] := 012;  waveArray[57] := 007;
   waveArray[58] := 004;  waveArray[59] := 002;
   waveArray[60] := 002;  waveArray[61] := 002;
   (* make wave *)
   waveName    := "Flute";
   waveBuffLen := HIGH(waveArray) + 1;
   waveBuffPtr := MakeWaveFromArray(waveArray);
END MakeFlute;


PROCEDURE SelectWaveForm(char: CHAR);

BEGIN
   (* release old wave buffer *)
   ReleaseWaveBuffer(waveBuffPtr, waveBuffLen);
   (* get new waveform *)
   CASE CAP(char) OF
     "W":
       WriteString("Wave File Name? ");
       ReadString(waveName);
       WriteString("\n");
       IF NOT WaveRead(
                  waveName, waveBuffPtr, waveBuffLen
              ) THEN
         waveBuffPtr := NIL;
         waveBuffLen := 0D;
       END (*IF*);
     |
     "1":
       waveName    := "Sine";
       waveBuffLen := 20D;
       waveBuffPtr := MakeSineWave(waveBuffLen);
     |
```

```
     "2":
       waveName    := "Sine32";
       waveBuffLen := 32D;
       waveBuffPtr := MakeSineWave(waveBuffLen);
     |
     "3":
       waveName    := "Sine64";
       waveBuffLen := 64D;
       waveBuffPtr := MakeSineWave(waveBuffLen);
     |
     "4":
       waveName    := "Triangle";
       waveBuffLen := 32D;
       waveBuffPtr := MakeTriangleWave(waveBuffLen);
     |
     "5":
       MakeFlute();
     |
     "6":
       waveName    := "Square";
       waveBuffLen := 20D;
       waveBuffPtr := MakeSquareWave(waveBuffLen);
     |
     "7":
       MakeClarinet();
     |
     "8":
       waveName    := "Sawtooth";
       waveBuffLen := 32D;
       waveBuffPtr := MakeSawtoothWave(waveBuffLen);
     |
     "9":
       waveName    := "Noise";
       waveBuffLen := 4096D;
       WriteString("Making Noise\n");
       waveBuffPtr := MakeNoiseWave(waveBuffLen);
     |
   ELSE
   END (*CASE*);
   (* attach wave to voices *)
   IF waveBuffPtr = NIL THEN
     WriteString("Error Creating Wave\n");
   ELSE
     SetVoiceWave(0, waveBuffPtr, waveBuffLen);
     SetVoiceWave(1, waveBuffPtr, waveBuffLen);
     SetVoiceWave(2, waveBuffPtr, waveBuffLen);
     SetVoiceWave(3, waveBuffPtr, waveBuffLen);
   END (*IF*);
   (* play sample note *)
   (* IF CAP(char) = "L" THEN *)
   IF waveBuffLen > 218D THEN
     PlayVoice(0, sampleFrequency, 0.0, sampleVolume);
   ELSE
     PlayVoice(0, 220.0, 0.0, sampleVolume);
   END (*IF*);
   Pause(2.0);
   StopVoice(0);
END SelectWaveForm;


PROCEDURE PlaySample();

VAR
   period   : CARDINAL;

BEGIN
   WriteString("Play Sample\n");
   (* calculate period *)
   IF PeriodCalculated(
        sampleFrequency,
        CARDINAL(waveBuffLen),
        period
      ) THEN
   END (*IF*);
   (* play *)
   PlayVoicePerCyc(0, period, sampleCycles, sampleVolume);
END PlaySample;


PROCEDURE ProcessCommands();

VAR
   char: CHAR;

BEGIN
   char := " ";
   LOOP
     IF char # LF THEN
       WriteString("==============================================");
       WriteString(" AudioProbe ");
       WriteString("==============================================");
       WriteString("\n");
       WriteString("\n");
       WriteString("WaveForm: ");
       WriteString(waveName);
       WriteString("\n");
       WriteString("\n");
       WriteString("Frequency: ");
       WriteReal(frequency, 1);
       WriteString("\n");
       ShowEnvelopeParameters(
         freqModAmt,
         freqAttackTime,
         freqDecayTime,
         freqSustainLevel,
         freqSustainTime,
         freqReleaseTime
       );
```

```
                WriteString("\n");
                WriteString("Volume: ");
                WriteCard(volume, 1);
                WriteString("\n");
                ShowEnvelopeParameters(
                    volModAmt,
                    volAttackTime,
                    volDecayTime,
                    volSustainLevel,
                    volSustainTime,
                    volReleaseTime
                );
                WriteString("\n");
                WriteString("Sample:");
                WriteString("  Frequency: ");
                WriteReal(sampleFrequency, 1);
                WriteString("  Volume: ");
                WriteCard(sampleVolume, 1);
                WriteString("  Cycles: ");
                WriteCard(sampleCycles, 1);
                WriteString("\n");
                WriteString("\n");
                WriteString("W-Load WaveForm  ");
                WriteString("1-Sine    ");
                WriteString("2-Sine32  ");
                WriteString("3-Sine64  ");
                WriteString("4-Triangle  ");
                WriteString("\n");
                WriteString("5-Flute  ");
                WriteString("6-Square  ");
                WriteString("7-Clarinet  ");
                WriteString("8-Sawtooth  ");
                WriteString("9-Noise  ");
                WriteString("\n");
                WriteString("F-Set Frequency  ");
                WriteString("V-Set Volume  ");
                WriteString("E-Frequency Envelope  ");
                WriteString("G-Volume Envelope  ");
                WriteString("\n");
                WriteString("P-Set Sample Frequency  ");
                WriteString("L-Set Sample Volume  ");
                WriteString("T-Set Sample Cycles  ");
                WriteString("\n");
                WriteString("B-Play Broken Chord  ");
                WriteString("C-Play Chord  ");
                WriteString("N-Play Note  ");
                WriteString("\n");
                WriteString("S-Play Sample  ");
                WriteString("Z-Play Siren  ");
                WriteString("Q-Quit   ");
                WriteString("\n");
            END (*IF*);
            (* accept and process command *)
            Read(char);
            CASE CAP(char) OF
                "B":
                    PlayBrokenChord();
                |
                "C":
                    PlayChord();
                |
                "E":
                    SetEnvelopeParameters(
                        "Frequency Envelope",
                        freqModAmt,
                        freqAttackTime,
                        freqDecayTime,
                        freqSustainLevel,
                        freqSustainTime,
                        freqReleaseTime
                    );
                |
                "F":
                    WriteString("Frequency? ");
                    ReadReal(frequency);
                    WriteString("\n");
                |
                "G":
                    SetEnvelopeParameters(
                        "Volume Envelope",
                        volModAmt,
                        volAttackTime,
                        volDecayTime,
                        volSustainLevel,
                        volSustainTime,
                        volReleaseTime
                    );
                |
                "L":
                    WriteString("Sample Volume? ");
                    ReadCard(sampleVolume);
                    WriteString("\n");
                    IF sampleVolume > 64 THEN
                        sampleVolume := 64;
                    END (*IF*);
                |
                "N":
                    PlayNote();
                |
                "P":
                    WriteString("Sample Frequency? ");
                    ReadReal(sampleFrequency);
                    WriteString("\n");
                |
                "Q":
                    WriteString("Quit\n");
                    EXIT (*LOOP*);
                |
                "S":
                    PlaySample();
                |
                "T":
                    WriteString("Sample Cycles? ");
                    ReadCard(sampleCycles);
                    WriteString("\n");
                |
                "V":
                    WriteString("Volume? ");
                    ReadCard(volume);
                    WriteString("\n");
                |
                "W", "1".."9":
                    SelectWaveForm(char);
                |
                "Z":
                    PlaySiren();
                |
            ELSE
            END (*CASE*);
        END (*LOOP*);
    END ProcessCommands;


VAR
    idx: CARDINAL;

BEGIN (* AudioProbe1 *)
    (* initialize wave buffer *)
    waveBuffPtr := NIL;
    waveBuffLen := 0D;
    (* initialize controls *)
    frequency        := 440.0;
    freqModAmt       := 0;
    freqAttackTime   := 0.5;
    freqDecayTime    := 0.25;
    freqSustainLevel := 75.0;
    freqSustainTime  := 1.0;
    freqReleaseTime  := 0.5;
    volume           := 0;
    volModAmt        := 100;
    volAttackTime    := 0.5;
    volDecayTime     := 0.25;
    volSustainLevel  := 75.0;
    volSustainTime   := 1.0;
    volReleaseTime   := 0.5;
    sampleFrequency  := 440.0;
    sampleVolume     := 32;
    sampleCycles     := 1;
    (* initialize envelopes *)
    FOR idx := 0 TO 3 DO
        SetVolumeEnvelope(
            idx,
            FALSE,
            volModAmt,
            volAttackTime,
            volDecayTime,
            volSustainLevel,
            volSustainTime,
            volReleaseTime
        );
        SetFrequencyEnvelope(
            idx,
            FALSE,
            freqModAmt,
            freqAttackTime,
            freqDecayTime,
            freqSustainLevel,
            freqSustainTime,
            freqReleaseTime
        );
    END (*FOR*);
    (* allocate voices *)
    IF VoicesAllocated() THEN
        (* set initial voice *)
        waveName    := "Sine";
        waveBuffLen := 20D;
        waveBuffPtr := MakeSineWave(waveBuffLen);
        IF waveBuffPtr # NIL THEN
            SetVoiceWave(0, waveBuffPtr, waveBuffLen);
            SetVoiceWave(1, waveBuffPtr, waveBuffLen);
            SetVoiceWave(2, waveBuffPtr, waveBuffLen);
            SetVoiceWave(3, waveBuffPtr, waveBuffLen);
            (* activate timer *)
            IF EnvelopeTimerCreated() THEN
                (* read and process commands *)
                ProcessCommands();
                (* release timer *)
                DeleteEnvelopeTimer();
            END (*IF*);
            (* release voices *)
            ReleaseVoices();
            (* release wave buffer *)
            ReleaseWaveBuffer(waveBuffPtr, waveBuffLen);
        END (*IF*);
    END (*IF*);
END AudioProbe1.
```

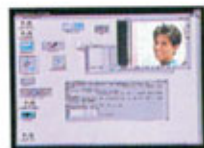*(Remaining AudioProbe listings are on disk!—Ed)*